

# **Yobotics!**

## **Simulation Construction Set**

**Copyright © 2000-2010 Yobotics, Inc. All rights reserved.  
Last modified May 5, 2010**



## Overview

The Yobotics Simulation Construction Set is a full-featured software package for easily and quickly creating simulations of mechanical devices, biomechanical systems, and robots. With it you can:

- Accurately and quickly (Order N) simulate rigid body physics.
- Access all joint positions, velocities, and torques.
- Define ground contour and ground contact models.
- Plot real-time graphs of any variable.
- Playback and rewind your simulations.
- Record and save data.
- Easily generate 3D graphics with texture mapping and camera controls.
- Import graphics from 3ds and vrmf file formats.
- Modify parameters as your simulation is running.
- Expand your simulations using the Java based Application Programmers Interface (API).
- Output JPG images and QuickTime Movies with the click of a button.
- Get Technical Support from both simulation and robot control experts at Yobotics.

The Simulation Construction Set is easy to use, yet powerful enough for creating complex simulations of robotic devices. Simulations of multi-jointed devices can be created in a matter of minutes. Arbitrary control can be added to these devices as each degree of freedom automatically has a simulated actuator associated with it. For power users, the simulations are easily extensible, as they are implemented 100% in Java, with a documented Application Programmers Interface (API).

With the Simulation Construction Set, you can be on your way to simulating complex robotic systems without going through the tedious and error prone process of manually calculating equations of motion. You'll get immediate instantaneous feedback through both cartoon animations of your simulation, and through plots of any simulation variable. You'll have the power to quickly tweak your robot designs and control systems by the easy modification of your simulations. Whatever your robotic simulation needs, the Yobotics! Simulation Construction Set will enable you to more quickly and accurately simulate your robot, allowing you to concentrate on the more important aspects of design and control.

This manual will explain how to setup and run the Yobotics! Simulation Construction Set. Several Tutorials are included for helping you climb the learning curve. After completing these tutorials you should be able to easily simulate complex robot devices, including walking robots, robot arms, mechanical links, and mechanisms. The manual includes documentation for the Simulation Construction Set API. If you have any questions or comments, contact [support@yobotics.com](mailto:support@yobotics.com).

## Getting Started

Since the Simulation Construction Set is written in 100% Java, it can be run from many platforms. We currently support Windows and Linux platforms. If you wish to run the Simulation Construction Set from a different platform, contact us and we will help you out. We recommend running the program on a Pentium 4 PC or better with 3D graphics card running either Windows 98, 2000, ME, NT, XP, Windows 7, or Linux. Instructions are provided below for installing the program on a Windows machine with a 3D graphics card supporting Open-GL or DirectX.

In this document, we will assume that you are using JBuilder2008 (which is based on Eclipse) from Embarcadero Code Gear to compile your simulations. However, any Java development tools may be used. If you need help with using a different development tool than JBuilder2008, contact [support@yobotics.com](mailto:support@yobotics.com).

### Windows Installation:




1. If you do not have a Java IDE, download and Install JBuilder2008R2. This is a Java development program that can be downloaded from <http://www.embarcadero.com>. They have a free download there and the basic version is free. Note that JBuilder2008 is Eclipse plus some nice plugins. The following instructions probably work for Eclipse as it is, but have not been tested on it yet.
2. If you are not familiar with your Java IDE, write a couple sample programs like “Hello World” to get familiar with using the IDE. Try to get comfortable with running a program and setting up libraries. In JBuilder2008, on the menu, click on Help->Help Contents. In the window that pops up, click on Java development user guide. This guide gives good information on creating projects, running them, and setting up libraries.
3. Unzip SimulationConstructionSet\_XXXXXX.zip onto your hard drive. Note that XXXXXX is the version number. It should create the directory SimulationConstructionSet and several sub directories.
4. If a previous version of the Simulation Construction Set is already installed, first remove it. To do so, search for any occurrences of SimulationConstructionSet.jar and IHMCUtilities.jar on your hard drive and delete them.
5. Download and install the latest version of Apple QuickTime from [www.apple.com](http://www.apple.com). This is necessary for creating QuickTime movies of your simulations.
6. Download and install the latest version of Java3D from Sun/Oracle. The easiest way to find it is usually to Google for “Java3d download”. As of May, 2010, the most recent version was 1.5.2. After installing, make sure you can see Java3D applets in your web browser. To check, Google for “Java3d applet examples” and try one of the many examples that show up.
7. In JBuilder2008R2 open up one of the example simulations in the SimulationConstructionSet/ExampleSimulations directory. There are several ways to do this. One is below.
  - a. From the JBuilder2008R2 menu, select, File->New->Java Project. Click on the “Create project from existing source” radio button. Browse to the source directory for the simulation. Continue through the wizard to create the project.

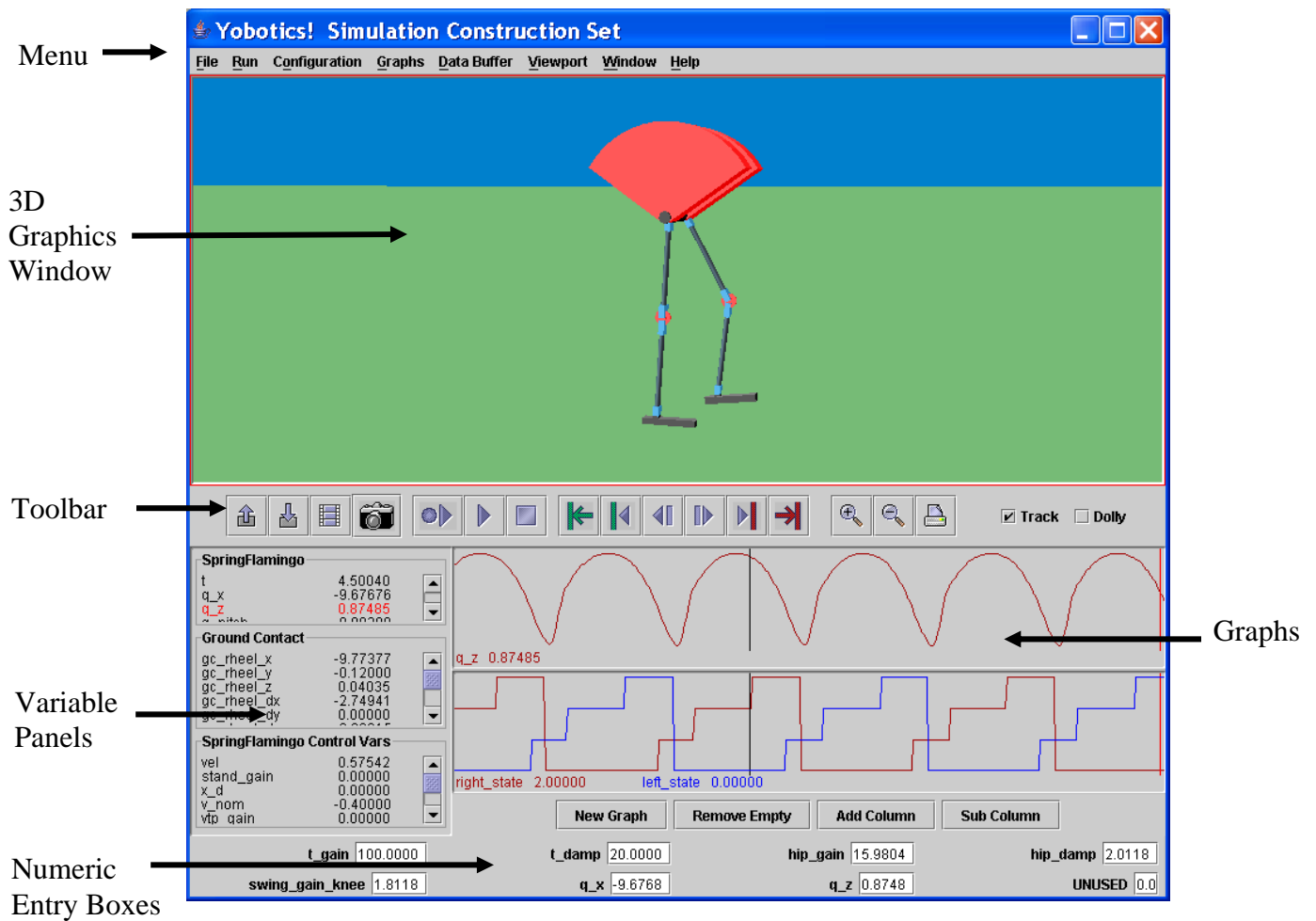
8. Add the jar files in SimulationConstructionSet/lib/ext to your Java classpath. There are several possible ways to do this:
  - a. Add all the files in SimulationConstructionSet/lib/ext to the lib/ext directory of your JDK. For JBuilder2008R2, it should be located at C:\JBuilder2008R2\jre\jre\lib\ext
  - b. In JBuilder2008, create a project. Then go to Project->Properties on the menu bar. Select Java Build Path on the left. Click on the Libraries tab. Click on Add external jars. Browse to the jar files in SimulationConstructionSet/lib/ext and include them all. This will need to be done for each project.
  - c. In JBuilder2008, do as in b) above, but create a User Library that contains all of the .jar files. Then this user library can be used for each project.
  - d. Add them to your classpath environment variable. This solution can be complicated and should be done by experienced computer gurus.
9. Add the Java3D files to your Java classpath. Do this in the same way as the above jar files, except that the Java3D files will be located somewhere like C:\Program Files\Java\Java3D\1.5.2\lib\ext. There should be j3dcore.jar, j3dutils.jar, and vecmath.jar.
10. Compile and run the example simulation. If you have any problems, the most likely problems are:
  - a. Not having the .jar files in the Java classpath. How to do this properly takes a bit of getting used to. If you are not a Java expert and have trouble with this step, we advise finding someone who is a Java expert to help you through it. Java is an extremely intuitive and easy to use language, except for setting up classpaths. Part of the blame is on the poor IDE design. Perhaps future IDEs will make this process easier...
  - b. Not having enough memory in the Java heap size. To fix this, add -Xmx1024m to the VM arguments in the Run Configurations window in JBuilder. This will tell the Java virtual machine to allow the heap space to grow up to 1024 MB.
11. Download the newest driver for your graphics card if you experience any difficulty with the 3D views. Several users have experience graphics problems in which the arrow tends to flash and the simulation is jerky. These problems seem to be fixed when the newest driver for their graphics card is installed.

Upon running the program, a Graphical User Interface (GUI) displaying the robot should appear as in **Figure 1**. Experiment with the simulation. Move on to the next section for instructions on running the simulation.

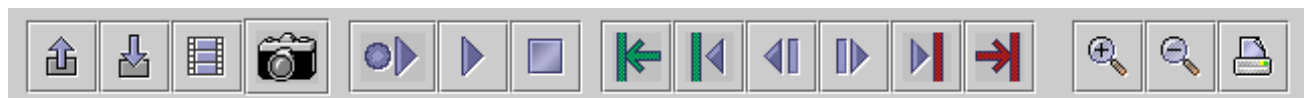
## Tutorial 1: Running a Simulation

Now that the software is installed, let's try running a simulation. The following tutorial will take you through some of the features of the simulation graphical user interface (GUI). The GUI provides the user with a 3D graphical image of the Simulation; it allows changing of variables using numerical entry boxes; it allows for the real-time plotting of variables; and it allows for the export and import of data, the creation of jpeg snapshots, and QuickTime movies.


1. Run the example simulation in the SimulationConstructionSet/ ExampleSimulations/ SpringFlamingo directory. A simulation of Spring Flamingo, a bipedal walking robot developed at the MIT Leg Laboratory, should appear. The simulation should resemble **Figure 1**.
2. Look around and familiarize yourself with the interface. A menu bar is on the top of the screen. A 3D graphical window showing the robot is below that. Below the 3D window is a toolbar with buttons for the most common functions. On the bottom left are variable panels containing lists of all the simulation and control variables. To the right of the variable panels are graph windows for graphing any variable. On the bottom are numeric entry boxes for changing the value of variables.
3. Start the simulation by either going to Simulate under the Run menu or by pressing the Simulate button  on the toolbar. The robot will start walking, the variable values in the variable panels will update, and the variable graphs will start filling with data.
4. After letting the simulation run for a few seconds, stop it by either going to Run->Stop on the menu bar or by pushing the stop button . The simulation should now stop.
5. Replay the simulated data by either going to Run->Play on the menu or by pushing the play button . The simulated data will be played back in the 3D graphics window in real time.
6. Change the camera view of the robot. To do so, first make sure that the Track or Dolly check boxes are unchecked. Then, place your mouse pointer over the 3D window and hold the left button down. Now, drag the mouse until the desired orientation is achieved. To zoom, hold the middle mouse button down while dragging the mouse up or down. To move the fix point of the camera (what the camera is looking at), hold down the right button while dragging the mouse. To set the fix point to be at the location of something in the graphical screen, hold down Shift and then click on the location on the screen. Play with moving the camera via the mouse until you are comfortable with it.
7. Change the camera parameters by using the dialog accessed through the pull-down menu (Camera->Camera Properties...). If tracking is enabled, then the camera will track the location specified by the variables  $q_x$ ,  $q_y$ ,  $q_z$ . If dolly is enabled, then the camera will move at an offset from the location specified by the variables  $q_x$ ,  $q_y$ ,  $q_z$ . Which variables are used for tracking or dolly can also be set using the SimulationConstructionSet API.












**Figure 1: Screen snapshot of Yobotics! Simulation Construction Set GUI Window with a Spring Flamingo Simulation.**

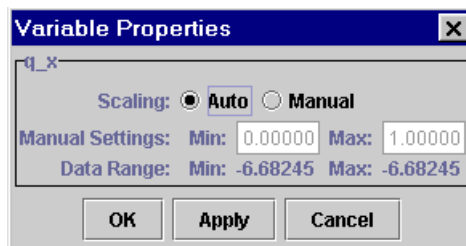


**Figure 2: Simulation GUI Toolbar.**

8. Stop the playback by either going to Run->Stop on the menu bar or by pushing the stop button .
9. Look at the graphs of the data. Notice that there is a green bar at the beginning of the data, a red bar at the end of the data, and a black bar somewhere in the middle. The green bar marks the data in point. The red bar marks the data out point. And the black bar marks the current index. The in and out points are used in any of the functions which require a window on the data. For example, when you played back the data, the playback looped to the in point once it reached the out point.

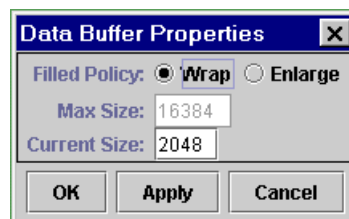
10. Zoom in on the data by going to Graphs->Zoom In on the menu or by pushing the Zoom In button . Push it a couple more times to zoom in further. Notice that zooming tends to center the data on the black line that marks the current index. Zoom out by going to Graphs->Zoom Out on the menu or by pushing the Zoom Out button . Zoom in and out until a few cycles of data fit the screen.
11. “Scrub” the data by left clicking on the data and dragging the mouse left and right. Notice that this moves the black line marking the current index. As you move the index, both the 3D graphics of the robot will be animated, and the variable values in the variable panels will update.
12. “Step” forward through the data, one point at a time, by either going to Run->Step Forward on the menu or by pushing the Step Forward button  or by pushing the right arrow key on your keyboard after clicking in the graph. “Step” backward through the data, one point at a time, by either going to Run->Step Backward on the menu or by pushing the Step Backward button  or by pushing the left arrow key on your keyboard after clicking in the graph.
13. “Pan” the data by first zooming into a section and then clicking on the data with the right mouse button and dragging left and right. This will move the data that you are zoomed into.
14. Zoom out so that you can see the in point (green line), out point (red line) and current index (black line) in the data. Go to the in point by either going to Run->Goto In Point on the menu or by pushing the Goto In Point button .
15. Go to the out point by either going to Run->Goto Out Point on the menu or by pushing the Goto Out Point button .
16. Now change the in point. First, left click somewhere in the middle of the data to move the current index there. Next, go to Run->Set In Point on the menu or push the Set In Point button .
17. Now change the out point. First, left click somewhere in the middle of the data to move the current index there. Next, go to Run->Set Out Point on the menu or push the Set In Point button .
18. Play the simulation (Run->Play on menu or play button ) and notice that it now loops around the new in and out points. Note that if the in point is after the out point, then the playback loops around the entire data buffer, which is a circular buffer.
19. Click on the “New Graph” button under the graphs. This will create a new empty graph.
20. Find  $q\_rh$  (the angle of the robot’s right hip) in the first variable panel on the left (labeled SpringFlamingo). Left click on it, which will highlight the variable. Middle click on the new empty graph (labeled “Click to graph selected variable.”) This will graph  $q\_rh$  in that graph. Each graph can contain up to 4 variables. Create some more new graphs, choose some more variables and place them in the graphs. Note that as more graphs are created, the others get smaller to accommodate them.
21. Notice that the name and current value of each variable is underneath the graph it appears in. Ungraph some of the variables by clicking their name with the middle mouse button. Note that if you remove all the variables from a graph, it will be empty but not disappear. If you wish to remove the empty graphs, click on the “Remove Empty” button.

22. Click on the “Add Column” button to increase the number of graph columns. Click on the “Sub Column” button to remove the number of graph columns.
23. Double click on a variable in one of the variable panels. A Variables Property Dialog Box (see Figure 3) will appear. This box will show that the variable is set to AutoScale. This means that when graphed, it will be plotted such that it vertically fills the graph, with the minimum value touching the bottom of the graph and the maximum value touching the top of the graph. Change to Manual scaling by clicking on the Manual radio button. Notice that the manual settings are now enabled. The default values are 0.0 minimum and 1.0 maximum. The data range is listed on the bottom. Change the minimum and maximum values. Graph the variable if it isn’t already (middle click in an open graph) and note that the variable is plotted with the new data range.









**Figure 3: Variable Properties Dialog.**

24. Double click on a graph containing a couple variables. A Variables Property Dialog Box (like the step above) will appear but will contain all the variables that are plotted in that graph. Set them to manual and change their plotting ranges. Click Apply to see the graphs change to reflect your changes. Click OK when done.
25. Double click on a graph containing two variables and change the plot type from ‘Time’ to ‘Phase’. The first variable will now be the x-axis, with the second variable the y-axis.
26. As the robot is simulated, the data is stored in a data buffer. The period in which data is stored defaults to once every 20 milliseconds (50 Hertz). This can be changed through the SimulationConstructionSet API. The default size of the data buffer is 2048 points. With these two defaults, 40.96 seconds of data can be recorded. If the simulation runs over this size, the buffer will become about 25% larger. It will continue to enlarge until it hits the max size limit that you set in the dialog box (default of 16384). Change these properties by going to Data Buffer->Data Buffer Properties on the menu. A dialog box pops up. Click on the Wrap radio button to change the fill policy from Enlarge to Wrap. With this policy, if you reach the end of the buffer, the current index is wrapped to the beginning of the buffer.



**Figure 4: Data Buffer Properties Dialog.**

27. The data buffer can be made smaller by cropping it to the in point and out point. Choose a window of data that you'd like to crop and set the in point to the beginning of the window  and the out point to the end of the window . Then go to Data Buffer->Crop Buffer to In/Out in the menu. The data should be cropped to the window you selected.
28. Sometimes you may wish to move the data so that the in point is the first index but you don't wish to crop the data. To do this, go to Data Buffer->Pack Buffer to In/Out on the menu. This will rotate the data in the buffer so that the in point is "packed" to the left.
29. Run the simulation for a little while and then set the in point and out point to create a window of data, which you wish to save. Go to File->Export Data on the menu or push the Export Data button . Select to save a compressed binary data file. A Save dialog box will appear. Choose the directory where you wish to save the data and enter a filename.
30. Reload the data by going to File->Import Data on the menu or by pushing the Import Data button . Choose the file that you just saved. The data will be loaded. Note that the buffer size will be the exact same size as the saved data. To change the buffer size or properties, see steps 22 and 23 above.
31. Save a snapshot of the 3D graphics image in jpeg format by going to File->Export Snapshot on the menu or by clicking on the Export Snapshot button . Enter the directory and name of the file. Make sure to end it with either a ".jpg" or ".jpeg" extension. Find the file and load it into an image viewer (double clicking on the file should do the trick) to see that it was saved.
32. Save a QuickTime movie of the simulation by going to File->Export Movie on the menu or by clicking on the Export Movie button . Enter the directory and name of the file. Make sure to end it with a ".mov" extension. Creating the movie may take a few minutes depending on how long of a window was selected with the in point and out point. After the movie is finished, find it and load it using the Apple QuickTime viewer (double clicking on the file should do the trick).
33. Change the value of a variable by first left-clicking on it in the variable panel to highlight it. Then click on the text "UNUSED" in the unused entry box on the bottom of the screen. The variable name should appear in the box along with the current variable value. To change the variable value, click in the box, enter a new value, and press Enter. To remove a variable from an entry box, right click on the variable name.
34. Go to Configuration on the Menu Bar. VarGroups, GraphGroups, and EntryBoxGroups have been configured using the SimulationConstructionSet API. Select a configuration and notice how the graphs change.
35. Go to Window->New Graph Window to create a new window for graphs. Then in that window select a configuration to plot some variables. The new graph window operates similarly to the graphs on the main window.
36. Select a new camera view by going to Viewport->Camera and selecting one of the available cameras.
37. Select a different viewport configuration by going to Viewport on the Menu Bar and selecting view2. The viewports were configured using the SimulationConstructionSet API. See the file SpringFlamingoSimulation.java for the code. With a multi-camera viewport, the active camera has a highlighted window. Any graphics or camera operations will occur on that camera.

38. Well, that's about it. Play with the simulation until you are comfortable with it. Then try some of the others. Trebuchet Demo is a simulation of a Trebuchet, a medieval siege warfare device. FallingBrick Demo is a falling object falling on a contoured surface. Mobile Demo is a mobile with 21 gimbal joints (36 degrees of freedom). To quit a simulation go to File->Exit.

## Tutorial 2: Creating a New Simulation

This tutorial assumes you know a little about JBuilder, and a little about Java. In parallel with these tutorials, you should also experiment with JBuilder to learn how to use its features. You should also learn Java, if you haven't already, by picking up a good Java book or two. We recommend Bruce Eckel's "Thinking in Java", which can be downloaded for free at [bruceeckel.com](http://bruceeckel.com), or purchased.

In this tutorial, we will create a simple simulation of a double pendulum. By the end of the tutorial you should be able to create and simulate simple passive systems with pin joints.

1. Start JBuilder2008R2.
2. Start a new project by going to File->New-> Java Project. The New Java Project wizard will appear for entering the project information. For project name enter "DoublePendulum". Select Create new project in workspace. Keep the other entries as their defaults. Click Next.
3. The Java Settings window of the project wizard will appear. Change the Default output folder to DoublePendulum/classes instead of DoublePendulum/bin.
4. Click on the Libraries tab and include the SimulationConstructionSet.jar, IHMCUtilities.jar, and the other jar files the same way you did for the previous simulation. Click finish when done.
5. Every simulation requires a Robot class and a Simulation class. Therefore we will create DoublePendulumRobot and DoublePendulumSimulation.
6. Add a new class by going to File->New->Class on the menu. For the Class Name enter "DoublePendulumSimulation". For the package enter com.yobotics.exampleSimulations.doublePendulum. Click Finish.
7. Fill in DoublePendulumSimulation as shown below:

```
package com.yobotics.exampleSimulations.doublePendulum;
import com.yobotics.simulationconstructionset.*;

public class DoublePendulumSimulation
{
    private SimulationConstructionSet sim;

    public DoublePendulumSimulation()
    {
        DoublePendulumRobot doublePendulum = new DoublePendulumRobot();

        sim = new SimulationConstructionSet(doublePendulum);
        sim.setGroundVisible(false);

        Thread myThread = new Thread(sim);
        myThread.start();
    }

    public static void main(String[] args)
    {
        DoublePendulumSimulation sim = new DoublePendulumSimulation();
    }
}
```

8. When your simulation is run, first the main method will be called. It creates a new DoublePendulumSimulation. In creating a DoublePendulumSimulation, a DoublePendulumRobot is first created, and then a SimulationConstructionSet object is created with that robot. A Thread is then created using the SimulationConstructionSet object. Finally the Thread is started, thereby starting your simulation. This simple template can be used for creating any simulation using the Simulation Construction Set.
9. Now create a DoublePendulumRobot by adding a new class (File->New Class...). For the class name, enter "DoublePendulumRobot". Click OK.
10. Fill in DoublePendulumRobot as shown below:

```

package com.yobotics.exampleSimulations.doublePendulum;

import com.yobotics.simulationconstructionset.*;
import javax.vecmath.*;

public class DoublePendulumRobot extends Robot
{
    private static final double
        L1 = 1.0, L2 = 2.0, M1 = 1.0, M2 = 1.0, R1 = 0.1, R2 = 0.05, Iyy1 = 0.5,
        Iyy2 = 1.0;

    public DoublePendulumRobot()
    {
        super("DoublePendulum");

        PinJoint pin1 = new PinJoint("joint1", new Vector3d(0.0, 0.0, 0.0), this,
Joint.Y);
        Link link1 = link1();
        pin1.setLink(link1);
        this.addRootJoint(pin1);

        Joint pin2 = new PinJoint("joint2", new Vector3d(0.0, 0.0, L1), this,
Joint.Y);
        Link link2 = link2();
        pin2.setLink(link2);
        pin1.addJoint(pin2);
    }

    private Link link1()
    {
        Link ret = new Link("link1");
        ret.setMass(M1);
        ret.setComOffset(0.0, 0.0, L1 / 2.0);
        ret.setMomentOfInertia(0.0, Iyy1, 0.0);

        LinkGraphics linkGraphics = new LinkGraphics();
        linkGraphics.addCylinder(L1, R1, YoAppearance.Red());

        ret.setLinkGraphics(linkGraphics);

        return ret;
    }
}

```

```

private Link link2()
{
    Link ret = new Link("link2");
    ret.setMass(M2);
    ret.setComOffset(0.0, 0.0, L2 / 2.0);
    ret.setMomentOfInertia(0.0, Iyy2, 0.0);

    LinkGraphics linkGraphics = new LinkGraphics();
    linkGraphics.addCylinder(L2, R2, YoAppearance.Green());

    ret.setLinkGraphics(linkGraphics);

    return ret;
}
}

```

11. Let's go through DoublePendulumRobot one line at a time:

- a. `package com.yobotics.exampleSimulations.doublePendulum;` This line states which package this file is in. See a Java reference book for more information on packages.
- b. `import com.yobotics.simulationconstructionset.*;` This line imports all the classes from the SimulationConstructionSet Library.
- c. `import javax.vecmath.*;` This imports the vector math library functions that come with the Java3d extension. In particular, we will use Vector3d in creating this robot.
- d. `public class DoublePendulumRobot extends Robot` This line declares the class DoublePendulumRobot to be a public class that extends Robot. The class Robot is included in the Simulation Construction Set and has built in graphics, dynamics, etc. Extending the class is an easy way to make a new type of robot, in this case a DoublePendulumRobot.
- e. `private static final double L1 = 1.0, L2 = 2.0, M1 = 1.0, M2 = 1.0, R1 = 0.1, R2 = 0.05, Iyy1 = 0.5, Iyy2 = 1.0;` This line defines the parameters of the robot. L1 and L2 are the link lengths, M1 and M2 are the link masses, and R1 and R2 are the radii of the links, Iyy1 and Iyy2 are the moments of inertia of the links. "private" means that these variables can only be accessed in this file. "static" means that there's only one copy of these variables, no matter how many DoublePendulumRobots are created, and "final" means that the values cannot be changed.
- f. `super("DoublePendulum");` This line creates an instance of the class Robot. The string "DoublePendulum" will be the name of the robot.
- g. `PinJoint joint1 = new PinJoint("joint1", new Vector3d(), this, Joint.Y);` This line creates a pin joint. The first parameter "joint1" is the name of the joint and will be used in all the variables associated with that joint. For instance `q_joint1` will be the joint angle, `qd_joint1` will be the joint velocity, and `tau_joint1` will be the joint torque. The second parameter "new Vector3d()" defines the offset of this joint from the previous joint. Since no offset is desired, the default vector (0,0,0) will be used. The third parameter "this" refers to the robot itself. The final parameter "Joint.Y" defines the axis of rotation for this pin joint.
- h. `Link link1 = link1();` This line creates a new link by calling the link1 method.

- i. `joint1.setLink(link1);` This line associates the Joint `joint1` with the Link `link1`.
  - j. `this.addRootJoint(joint1);` This line adds `joint1` as a `rootJoint` of the robot. In order to be part of a robot, a joint must either be added as a root joint, or be attached to a parent joint. This ensures the tree structure (forest structure if there are multiple root joints) of the robot.
  - k. `Joint joint2 = new PinJoint("joint2", new Vector3d(0.0,0.0,L1), this, Joint.Y);` This line creates the second pin joint in the same way that the first one was created. This time however, the offset vector is  $(0.0,0.0,L1)$  since the second pin joint should be placed a distance of  $L1$  in the  $Z$  direction from the previous joint.
  - l. `Link link2 = link2();` This line creates the second link by calling method `link2`.
  - m. `joint2.setLink(link2);` This line associates the Joint `joint2` with the Link `link2`.
  - n. `joint1.addJoint(joint2);` This line adds the Joint `joint2` as a child to the Joint `joint1`.
  - o. `private Link link1();` This line is the beginning of the method `link1` which will return a Link.
  - p. `Link ret = new Link("link1");` This line creates a new Link named "link1" and assigns it to the return value `ret`.
  - q. `ret.setMass(M1);` This line sets the mass of the link to  $M1$ .
  - r. `ret.setComOffset(0.0,0.0,L1/2.0);` This line sets the center of mass offset of the link to be  $L1/2$  in the  $Z$  direction.
  - s. `ret.setMomentOfInertia(0.0,Iyy1,0.0);` This line sets the moment of inertia. Note that the moment of inertia is defined *about the center of mass*. Therefore, if the moment of inertia is set to zero, the link will be a point mass.
  - t. `linkGraphics.addCylinder(L1,R1);` This line adds a cylinder of length  $L1$  and radius  $R1$ . This will make the link visually look like a cylinder in the graphical interface.
  - u. `return ret;` This line returns the link. The method `link2` is similar to `link1` and thus not reproduced here.
12. Before compiling, make sure that the Simulation Construction Set Libraries are referenced properly in JBuilder2008. Go to Project->Properties on the menu. Click on Java Build Path on the left. Select the Libraries tab. Make sure that all the required .jar files are included in the libraries.
  13. Open DoublePendulumSimulation and click on the green and white triangle in the top bar of buttons. Alternately, on the menu, select Run->Run As->Java Application. This should compile and run the project. If there are any problems, they should appear in the Problems Tab on the bottom of the JBuilder IDE.
  14. A simulation of the DoublePendulumRobot should appear. If there were any troubles in compiling.
  15. Start running the simulation. Note that the pendulum doesn't do anything as it is perfectly (and unrealistically) suspended straight up. Stop the simulation. Then put `q_joint1` in a numeric entry box. Change its value from 0.0 to 0.01.
  16. Run the simulation again. Now the pendulum will fall and flop around. Notice that its behavior is chaotic, as this is a chaotic system. However, even though it is a chaotic system, if run twice in a row, it will produce the exact same trajectories due to starting from exactly the same initial conditions. If start from slightly different conditions, it should produce different behavior.

17. Experiment with different lengths, masses, and center of mass locations until you are comfortable with this simulation and compiling simulations with JBuilder2008R2.

## Tutorial 3: Adding Control to a Simulation.

In this tutorial, we will apply a control algorithm for balancing the double pendulum in the previous example. We will treat the double pendulum as an “Acrobot” in which the first joint (shoulder) is a free bearing and thus can apply no torque. The second joint (elbow) will be controlled to balance the robot in an inverted position. The parameters we will use and the balancing controller are taken from the paper “The Swing Up Control Problem for the Acrobot” by Mark Spong. This paper can be retrieved from [http://robot0.ge.uiuc.edu/~spong/Conference\\_Proceedings/](http://robot0.ge.uiuc.edu/~spong/Conference_Proceedings/) Here we will only implement the balancing controller. The swing up control is left as an exercise.

1. Do the previous tutorial if you haven’t already. Open the DoublePendulum project from the previous tutorial in JBuilder.
2. Go to File->New Class... and create the class named DoublePendulumController in the package doublependulum.
3. Fill in DoublePendulumController as shown below:

```
package com.yobotics.exampleSimulations.doublePendulum;

import com.yobotics.simulationconstructionset.*;

public class DoublePendulumController implements RobotController
{
    private YoVariable var1, var2;

    private YoVariable tau_joint1, tau_joint2, q_joint1, q_joint2, qd_joint1,
qd_joint2;

    private final YoVariableRegistry registry = new
YoVariableRegistry("DoublePendulumController");
    private YoVariable k1, k2, k3, k4;

    public DoublePendulumController(DoublePendulumRobot rob)
    {
        q_joint1 = rob.getVariable("q_joint1");
        qd_joint1 = rob.getVariable("qd_joint1");
        tau_joint1 = rob.getVariable("tau_joint1");

        q_joint2 = rob.getVariable("q_joint2");
        qd_joint2 = rob.getVariable("qd_joint2");
        tau_joint2 = rob.getVariable("tau_joint2");

        k1 = new YoVariable("k1", registry);
        k1.val = -242.52;
        k2 = new YoVariable("k2", registry);
        k2.val = -96.33;
        k3 = new YoVariable("k3", registry);
        k3.val = -104.59;
        k4 = new YoVariable("k4", registry);
        k4.val = -49.05;
    }
}
```

```

public void doControl()
{
    tau_joint1.val = 0.0;
    tau_joint2.val = -k1.val * q_joint1.val - k2.val * q_joint2.val -
k3.val * qd_joint1.val - k4.val * qd_joint2.val;
}

public YoVariableRegistry getYoVariableRegistry()
{
    return registry;
}
}

```

4. Let's go through DoublePendulumController one line at a time:

- a. `package com.yobotics.exampleSimulations.doublePendulum;` This line states what package the class is in. See a Java reference book for more information on packages.
- b. `import com.yobotics.simulationconstructionset.*;` This line imports all the classes from the SimulationConstructionSet library.
- c. `public class DoublePendulumController implements RobotController` This line declares the class DoublePendulumController to be a public class that implements the interface RobotController. This interface is part of the Simulation Construction Set. To implement the interface, the class must contain the methods `getYoVariableRegistry()` and `doControl()`. Notice that these two methods are indeed defined for DoublePendulumController.
- d. `private YoVariable tau_joint1, tau_joint2, q_joint1, q_joint2, qd_joint1, qd_joint2;` These are some of the variables that are automatically created during the creation of the DoublePendulumRobot. In order to use them in the controller, they must be named and extracted from the robot. `tau_*` is the torque at the joint, `q_*` is the position of the joint, and `qd_*` is the velocity of the joint.
- e. `private YoVariable k1,k2,k3,k4;` These are the new control variables which will be used for control of the DoublePendulumRobot.
- f. `public DoublePendulumController(DoublePendulumRobot rob)` This is the constructor method for a DoublePendulumController. It requires passing the robot in, so that position and velocity variables can be extracted from it.
- g. `q_joint1 = rob.getVar("q_joint1");` This line, and the others like it, extract the named variable from the robot.
- h. `k1 = new YoVariable("k1", registry); k1.val = -242.52;` This line, and the others like it, create new variables for the control system and initialize their values. Note that to set or read the value of a variable, you must use the `.val` extension, or call the `set()` methods.
- i. `public YoVariableRegistry getYoVariableRegistry()` This method must be implemented to meet the requirements of the RobotControl interface. It returns the YoVariableRegistry that was used for registering all the YoVariables for the controller. This is important so that the rest of the Simulation Construction Set can access the newly created variables.

- j. `public void doControl()` This method must be implemented to meet the requirements of the `RobotControl` interface. It is where the control code is located. This control code gets called during every integration “tick”.
  - k. `tau_joint1.val = 0.0;` In the “Acrobot”, the first joint is a free bearing. Therefore, the torque of joint1 is set to zero to emulate a perfect bearing joint.
  - l. `tau_joint2.val = -k1.val * q_joint1.val - k2.val * q_joint2.val - k3.val * qd_joint1.val - k4.val * qd_joint2.val;` This is the balance control code. The torque at the second joint is a function of the positions and velocities of both the first and second joints. This control algorithm is sufficient to balance the Acrobat in the upright configuration under small perturbations.
5. In order for the control code to be accessed, it must be set in the Simulation Construction Set. Therefore, in `DoublePendulumSimulation`, after the line `DoublePendulumRobot doublePendulum = new DoublePendulumRobot();` add the lines
- ```
DoublePendulumController controller = new
    DoublePendulumController(doublePendulum);
doublePendulum.setController(controller);
```
- This will get the robot variables from the robot and create a new `DoublePendulumController` called `controller`. This controller will then be passed to the robot using `setController`, so that the Simulation Construction Set now has access to the control code.
6. In `DoublePendulumRobot`, check that the mass, length, and inertia parameters are set as follows:
- ```
private static final double L1 = 1.0, L2 = 2.0, M1 = 1.0, M2 = 1.0, R1 =
0.1, R2 = 0.05, Iyy1 = 0.5, Iyy2 = 1.0;
```
- These are the parameters used in Mark Spong’s paper “The Swing Up Control Problem for the Acrobot” and are what control parameters  $k_1$ ,  $k_2$ ,  $k_3$ , and  $k_4$  are tuned for.
- 7. Compile and run the project. If there were any troubles in compiling, JBuilder will let you know in the Problems tab and point you to where the bugs are.
  - 8. Start running the simulation. Note that the pendulum doesn’t do anything as it is perfectly (and unrealistically) suspended straight up. Stop the simulation. Then put `q_joint1` in a numeric entry box. Change its value from 0.0 to 0.01.
  - 9. Run the simulation again. Instead of falling down, this time the pendulum should balance by actuating its second joint. If it doesn’t balance, make sure that the mass and moment of inertia parameters are set as shown above, that the control parameters  $k_1$ ,  $k_2$ ,  $k_3$ , and  $k_4$  are set as shown above. Also, make sure that the controller is being run by plotting `tau_joint2` in a graphics window. `tau_joint2` should change as the joint angles and velocities change.
  - 10. Experiment with `q_joint1` and `q_joint2` to see how much of a perturbation the Acrobot can recover from. Once the perturbation is too large, the simple linear controller will go crazy. What is needed is a “swing up” controller.
  - 11. As an exercise, either implement Spong’s swing up controller from his paper or implement a simple open-loop swing up controller (a sinusoidal torque at joint2 should do the trick). Once the swing up controller swings the robot up, then trigger in the balancing controller. Note also that you may need to read the joint sensors ( $q_*$ ) modulo  $2\pi$ , in case the links wrap around. If you need to access time, the variable “`t`” is automatically created and can be accessed in the same way as the joint angle variables:

```
YoVariable t = rob.getVar("t");
```

12. Next we will describe the Link API (Application Programmers Interface). These are the classes and methods which are used for defining the geometry and inertial properties of the links of the robot simulations. Then, in the next tutorial, we will experiment with the various shapes which are available.

## Link API

The following tables list the Link constructors and methods. **Table 1** contains the constructor and methods for setting the mass, center of mass offset, and moment of inertia. These are the only methods which effect the dynamics of the link. All of the others are for graphical purposes only. The center of mass offset is the vector from the joint to the center of mass. The moment of inertia is defined about the center of mass. Thus a moment of inertia of zero corresponds to a point mass.

Table 2 lists the methods for adding rotation and translation. These transformations accumulate until identity() is called, in which the transformation for new shapes to be added is reset to the origin. Note that if you rotate and then translate, the translation will be along the new coordinate system defined by the rotation. For example,

```
rotate(Math.PI/2.0, Link.X); translate(0.0,1.0,0.0);
```

is equivalent to

```
translate(0.0,0.0,1.0); rotate(Math.PI/2.0, Link.X);
```

Table 3 lists the methods available for adding shapes. The first method addShape(Geometry, Appearance) is a lower-level interface for creating a shape using the given Geometry and Appearance. Geometry and Appearance are classes defined in the Java3D API. This function is for advanced users who need to make custom shapes. We recommend reading Java3D reference manuals and becoming familiar with the Java3D API before using this method. The other methods are easy to use interfaces for adding standard shapes such as cubes, spheres, cones, ellipses, etc. Each method can be called with or without an Appearance. If called without an Appearance, the shape will be colored black. In order to make creating different Appearances easy, we've added a class called YoAppearance which contains static methods for creating common colors and appearances. Table 4 lists the methods in the YoAppearance helper API. To add a shape to a link, you simply use the corresponding method. For example, the following code will create a red sphere with radius of 0.3 meters and of the given mass and moment of inertia.

```
Link link1 = new Link("link1");
link1.setMass(1.0);
link1.setMomentOfIntertia(0.1, 0.1, 0.1);
link1.setComOffset(0.0,0.0,0.0);
link1.addSphere(0.3, YoAppearance.Red());
```

**Table 1: Creating a Link and Setting its Properties**

Method or Constructor	Purpose
Link(String lname)	Creates a link with name lname.
void setMass(double mass)	Sets the mass of the link.
void setMomentOfIntertia(double Ixx, double Iyy, double Izz)	Sets the moment of inertia of the link about the center of mass.
void setComOffset(double xOffset, double yOffset, double zOffset)	Sets the center of mass offset of the link with respect to its corresponding joint.

**Table 2: Translating and Rotating the Reference Frame for the Next Shape**

Method or Constructor	Purpose
void identity()	Reset back to the joint origin.
void translate(double tx, double ty, double tz)	Translate by (tx,ty,tz)
void rotate(double rotAng, int rotAxis)	Rotate by rotAng about rotAxis. rotAxis can be Link.X, Link.Y, or Link.Z
public static final int Link.X; public static final int Link.Y; public static final int Link.Z;	Constants for specifying axis of rotation for Link.rotate method.

**Table 3: Adding Shapes to the Link**

Method or Constructor	Purpose
void addShape(Geometry, Appearance)	Adds a shape of the given Geometry and Appearance. Geometry and Appearance are classes in the Java3D API. Refer to a Java3D reference manual for more information.
void addCube(double lx, double ly, double lz) void addCube(double lx, double ly, double lz, Appearance)	Adds a cube with lengths lx, ly, lz. Origin is at the center of the base of the cube (not at the center of mass of the cube)
void addWedge(double lx, double ly, double lz) void addWedge(double lx, double ly, double lz, Appearance)	Adds a wedge (a cube cut diagonally in half) with lengths lx, ly, lz. Origin is at the center of the base of the wedge.
void addSphere(double radius) void addSphere(double radius, Appearance)	Adds a sphere of the given radius. Origin is at the center of the sphere.
public void addEllipsoid(double xRad, double yRad, double zRad) public void addEllipsoid(double xRad, double yRad, double zRad, Appearance)	Adds an ellipsoid (sphere with 3 different radii). Origin is at the center of the ellipsoid.
public void addHemiEllipsoid(double xRad, double yRad, double zRad) public void addHemiEllipsoid(double xRad, double yRad, double zRad, Appearance)	Adds a hemi-ellipsoid (top half of an ellipsoid). The radii of the base is xRad and yRad. The height is zRad. Origin is at the center of the base of the hemi-ellipsoid.
void addCylinder(double height, double radius) void addCylinder(double height, double radius, Appearance)	Adds a cylinder with the given height and radius. Origin is at the center of the base of the cylinder (not at the center of mass of the cylinder)
void addCone(double height, double radius) void addCone(double height, double radius Appearance)	Adds a cone of given height and radius. Origin is at the center of the base of the cone.
void addGenTruncatedCone(double height, double bx, double by, double tx, double ty) void addGenTruncatedCone(double height, double bx, double by, double tx, double ty, Appearance)	Adds a general truncated cone with the given height. The base is an ellipse with radii of bx and by. The top is an ellipse with radii of tx and ty. Origin is at the center of the base of the cone.

<pre>void addArcTorus(double startAngle,     double endAngle,     double majorRadius,     double minorRadius) void addArcTorus(double startAngle,     double endAngle,     double majorRadius,     double minorRadius,     Appearance)</pre>	<p>Adds a section of a torus, from startAngle to endAngle. The majorRadius is the distance from the center to the center of the torus. The minor radius is the radius of the torus itself. Origin is at the center of the torus.</p>
<pre>void addPyramidCube(double lx, double ly, double lz,     double lh) void addPyramidCube(double lx, double ly, double lz,     double lh, Appearance)</pre>	<p>Adds a 12 sided cube with pyramids on both ends. The cube has sides of length lx,ly, and lz. The pyramids are each of height lh. Origin is at the center of mass.</p>
<pre>void add3DSFile(String fileName) void add3DSFile(String fileName, Appearance app) add3DSFile(URL fileURL, Appearance app)</pre>	<p>Adds the geometry specified in the 3D Studio Max (3ds) file. Includes the texture mapping, but requires the texture files to be located with the 3ds file.</p>
<pre>void addVRMLFile(URL fileURL) void addVRMLFile(URL fileURL, Appearance)</pre>	<p>Adds the geometry specified in the VRML file. Sets the Appearance to that specified. If no Appearance is specified, then uses the Appearance in the VRML file. Note: Only works for VRML Version 2.0 and not Version 1.0</p>
<pre>void addCoordinateSystem (double length)</pre>	<p>Adds a coordinate system with rods of the given length. The x axis is red, the y axis is white, and the z axis is blue.</p>

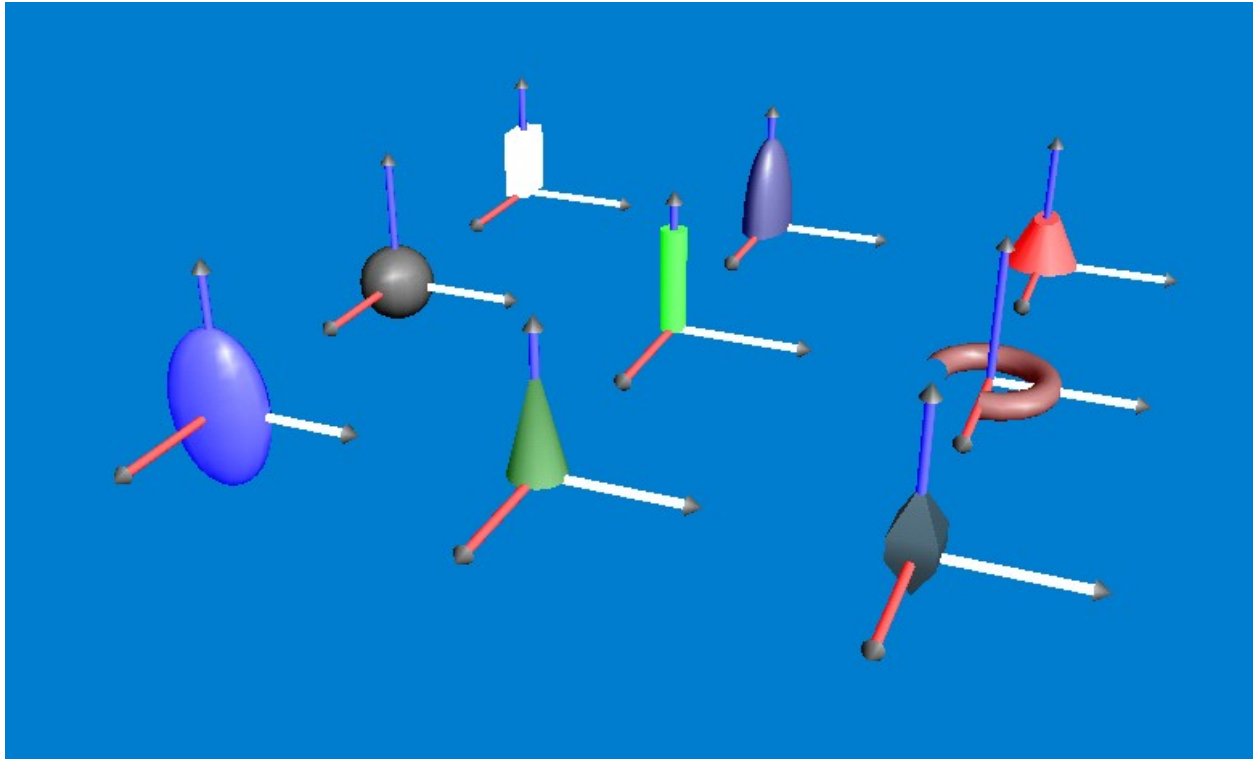
**Table 4: YoAppearance API. These static functions return Appearances with various colors or material looks.**

Method or Constructor	Purpose
Appearance YoAppearance.RGBColor(float red, float green, float blue)	Returns an Appearance with the given RGB values, each between 0.0 and 1.0
Appearance YoAppearance.Black()	Returns a black Appearance.
Appearance YoAppearance.White()	Returns a white Appearance.
Appearance YoAppearance.Blue()	Returns a blue Appearance.
Appearance YoAppearance.DarkBlue()	Returns a dark blue Appearance.
Appearance YoAppearance.Red()	Returns a red Appearance.
Appearance YoAppearance.DarkRed()	Returns a dark red Appearance.
Appearance YoAppearance.Green()	Returns a green Appearance.
Appearance YoAppearance.DarkGreen()	Returns a dark green Appearance.
Appearance YoAppearance.Silver()	Returns a silver Appearance.
Appearance YoAppearance.Gray()	Returns a gray Appearance.
Appearance YoAppearance.Maroon()	Returns a maroon Appearance.
Appearance YoAppearance.Purple()	Returns a purple Appearance.
Appearance YoAppearance.Fuchsia()	Returns a fuchsia Appearance.
Appearance YoAppearance.Olive()	Returns an olive Appearance.
Appearance YoAppearance.Yellow()	Returns a yellow Appearance.
Appearance YoAppearance.Navy()	Returns a navy Appearance.
Appearance YoAppearance.Teal()	Returns a teal Appearance.

Appearance YoAppearance.Aqua()	Returns a aqua Appearance.
Appearance YoAppearance.BlackMetalMaterial();	Returns a black metal Appearance.
Appearance YoAppearance.AluminumMaterial();	Returns a shiny aluminum Appearance.
void YoAppearance.makeTransparent(Appearance app, float transparency);	Sets the transparency of Appearance app to the desired value. 1.0 is fully transparent.
void makeLineDrawing(Appearance app);	Sets the polygon attributes of Appearance app such that only edges are drawn.
Appearance YoAppearance.EarthTexture(Component comp);	Returns a texture mapped Earth Appearance. The Earth graphics are taken from the Java3D tutorials.
Appearance YoAppearance.StoneTexture(Component comp);	Returns a texture mapped stone Appearance. The stone graphics are taken from the Java3D tutorials.

## Tutorial 4: Creating Links

In this tutorial, we will create various shapes and add them to a link. Figure 5 shows nine different shapes, each with a coordinate system at their origin point.



**Figure 5: Nine example shapes. Coordinate systems are located at the origin of each shape.**

1. Create a new project in JBuilder2008R2 with the name LinkExamples.
2. Add the class LinkExamplesSimulation
3. Fill in LinkExamplesSimulation as follows:

```
package linkexamples;
import com.yobotics.simulationconstructionset.*;

public class LinkExamplesSimulation
{
    private SimulationConstructionSet sim;

    private static final double CUBE_L = 0.2,
        CUBE_W = 0.1,
        CUBE_H = 0.3;
    private static final double SPHERE_R = 0.15;
    private static final double ELLIPSOID_RX = 0.1,
        ELLIPSOID_RY = 0.2,
        ELLIPSOID_RZ = 0.3;
    private static final double HEMI_ELLIPSOID_RX = 0.2,
```

```

        HEMI_ELLIPSOID_RY = 0.1,
        HEMI_ELLIPSOID_RZ = 0.4;
private static final double CYLINDER_H = 0.4, CYLINDER_R = 0.05;
private static final double CONE_H = 0.4, CONE_R = 0.1;
private static final double GEN_TRUNCATED_CONE_HEIGHT = 0.2,
        GEN_TRUNCATED_CONE_BX = 0.15,
        GEN_TRUNCATED_CONE_BY = 0.15,
        GEN_TRUNCATED_CONE_TX = 0.1,
        GEN_TRUNCATED_CONE_TY = 0.05;
private static final double ARC_TORUS_START_ANG = 0.0,
        ARC_TORUS_END_ANG = 1.5*Math.PI,
        ARC_TORUS_MAJ_RAD = 0.2,
        ARC_TORUS_MIN_RAD = 0.05;
private static final double PYRAMID_CUBE_LX = 0.15,
        PYRAMID_CUBE_LY = 0.15,
        PYRAMID_CUBE_LZ = 0.08,
        PYRAMID_CUBE_LH = 0.15;

private static final double OFFSET = 1.2, COORD_LENGTH = 0.5;

public LinkExamplesSimulation()
{
    Robot nullBot = null;
    sim = new SimulationConstructionSet(nullBot);

    Link exampleShapes = exampleShapes();
    sim.addStaticLink(exampleShapes);

    Thread myThread = new Thread(sim);
    myThread.start();
}

public static void main(String[] args)
{
    LinkExamplesSimulation sim = new LinkExamplesSimulation();
}

private Link exampleShapes()
{
    Link ret = new Link("example shapes");

    // Cube
    ret.addCoordinateSystem(COORD_LENGTH);
    ret.addCube(CUBE_L, CUBE_W, CUBE_H);

    // Sphere
    ret.translate(OFFSET, 0.0, 0.0);
    ret.addCoordinateSystem(COORD_LENGTH);
    ret.addSphere(SPHERE_R);

    // Ellipsoid
    ret.translate(OFFSET, 0.0, 0.0);
    ret.addCoordinateSystem(COORD_LENGTH);
}

```

```

ret.addEllipsoid(ELLIPSOID_RX, ELLIPSOID_RY, ELLIPSOID_RZ);

// Hemiellipsoid
ret.translate(-2.0*OFFSET,OFFSET,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addHemiEllipsoid(HEMI_ELLIPSOID_RX, HEMI_ELLIPSOID_RY,
                    HEMI_ELLIPSOID_RZ);

// Cylinder
ret.translate(OFFSET,0.0,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addCylinder(CYLINDER_H, CYLINDER_R);

// Cone
ret.translate(OFFSET,0.0,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addCone(CONE_H, CONE_R);

//GenTruncatedCone
ret.translate(-2.0*OFFSET,OFFSET,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addGenTruncatedCone(GEN_TRUNCATED_CONE_HEIGHT,
                       GEN_TRUNCATED_CONE_BX, GEN_TRUNCATED_CONE_BY,
                       GEN_TRUNCATED_CONE_TX, GEN_TRUNCATED_CONE_TY);

// ArcTorus
ret.translate(OFFSET,0.0,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addArcTorus(ARC_TORUS_START_ANG, ARC_TORUS_END_ANG,
               ARC_TORUS_MAJ_RAD, ARC_TORUS_MIN_RAD);

// PyramidCube
ret.translate(OFFSET,0.0,0.0);
ret.addCoordinateSystem(COORD_LENGTH);
ret.addPyramidCube(PYRAMID_CUBE_LX, PYRAMID_CUBE_LY, PYRAMID_CUBE_LZ,
                  PYRAMID_CUBE_LH);

return ret;
}
}

```

4. The structure of this code should now be familiar to you. Notice, however, that this is purely a static graphic we are creating. There is no robot, and thus no dynamics. This will allow us to focus on creating the graphics of a link.
5. Run `LinkExampleSimulation`. You should get a view resembling that of Figure 5.
6. Modify the values of the parameters of the shape and notice how it changes the shapes.
7. Notice the location of the coordinate systems relative to each shape. The origin of each coordinate system is at the origin of each shape.
8. Add rotations and more translation and notice their effects.
9. Notice that all of the shapes are black. Try creating some with different appearances using the `YoAppearance` utility API.
10. Once you are comfortable with the shapes and appearances, try experimenting to make an object, such as a snowman, out of the shapes.

## Robot and Joint API

A Robot consists of one or more trees of Joints, with each Joint having a corresponding link with mass and inertia. For example, if you are creating a bipedal robot, the first Joint might be a FloatingJoint (6 degree of freedom joint) with its link set to be the torso of the robot. Children of the FloatingJoint would then be the 2 hips with Links set to be the upper legs, and two shoulder joints, with Links being the upper arms. The structure would continue in that fashion with knees being children of the hips, ankles being children of the knees, etc.

The following tables list the various Robot and Joint constructors and methods. Table 5 contains the constructors and methods for the class Robot. To create a Robot you first call its constructor with its name, then create a tree of joints, and then set the root joint of that tree using Robot.addRootJoint(Joint root). You may add multiple root joints using addRootJoint, if your Robot is comprised of several independent trees of Joints. If the robot has a control system or ground contact points, you set the controller using setController or set the GroundContactModel using setGroundContactModel. To change the value of a variable in your robot, use setVariable, or to extract a YoVariable from the robot, use getVar() with the name of the variable.

Table 6 lists the Joint constructors and the methods which are specific to the type of joint. The first two Joints, FloatingJoint and PlanarFloatingJoint, can not be children of any other joint and cannot apply torques. They are used for free bodies, such as the body of a walking robot. If the body is unconstrained (has 6 degrees of freedom – 3 translations and 3 rotations), then use a FloatingJoint. If the body is confined to a plane (2 translations and 1 rotation perpendicular to the 2 translations), then use a FloatingPlanarJoint. The other types of joints are either PinJoint, which allows rotation, or SliderJoint, which allows translation, or some combination of PinJoints and SliderJoints. A CylinderJoint is a PinJoint and a SliderJoint, with rotation and translation both about the same axis. A UniversalJoint is 2 perpendicular PinJoints. A GimbalJoint consists of 3 perpendicular PinJoints.

In creating a Joint, you must specify the offset vector from the previous joint. This vector is defined when the rotations and translations of the previous joint are set to zero. For example, if a humanoid robot is created such that its arms hang straight down when the shoulders joint angles are zero, then the vector from the shoulder to the elbow will be (0.0,0.0,-UPPER\_ARM\_LENGTH). In creating a Joint, you must also specify the axes of rotation as one of Joint.X, Joint.Y, or Joint.Z. In the case of PinJoints and SliderJoints, you may also specify an axis of rotation by providing a Vector3d that defines the axis.

Most of the joints have the capability for setting damping, limit stops, and for setting their initial state. Damping is implemented as a linear viscous friction (torque is proportional to velocity). Limit stops are implemented as virtual spring-damper systems, with the given spring and damping constants.

Table 7 lists the methods which all Joints share. Each joint has a corresponding link. The link must be created as described in the Link API description from above. To set the link to the joint you use the setLink(Link l) method.

To add a child Joint to another Joint, use the addJoint(Joint nextJoint) method.

KinematicPoint, ExternalForcePoint and GroundContactPoint create various points that can be added using the corresponding methods. A KinematicPoint can be used to track the world coordinates and velocity of a point on the robot. ExternalForcePoint extends KinematicPoint and adds functionality for applying forces or impulses to the point. GroundContactPoint extends ExternalForcePoint and adds functionality for ground contact modeling. Each type of point is added to its parent Joint and moves with that joint.

**Table 5: Robot Constructor and Methods.**

Method or Constructor	Purpose
Robot(String name)	Creates a new robot.
void addRootJoint(Joint root)	Adds a root joint for this robot.
public void setGravity(double gZ) void setGravity(double gX, double gY, double gZ)	Sets gravity for this robot. Default is (0.0, 0.0, -9.81).
void setController(RobotController c)	Sets the RobotController for this robot.
void setGroundContactModel(GroundContactModel)	Sets the GroundContactModel for this robot.
ArrayList getGroundContactPoints()	Returns the ground contact points associated with this robot. They will be stored in an ArrayList, which is a Java utility class in (java.util.ArrayList)
void setVariable(String name, double val); YoVariable getVar(String name)	Gets or sets the variable with the given name.
void addStaticLink(Link staticLink) void addStaticBranchGroup(BranchGroup staticBranchGroup)	Adds a static Link with no physics, or a static Java3D BranchGroup. For creating the surrounding static world graphics.

**Table 6: Joint Constructors and Joint Specific Methods.**

Method or Constructor	Purpose
FloatingJoint(String jname, String varName, Vector3d offset, Robot rob) FloatingJoint(String jname, Vector3d offset, Robot rob)	Creates a 6 degree of freedom floating joint. This joint can only be the root joint for a robot. YoVariables will be automatically created and added to the robot. If no varName is given, these variables will be $q_{(x,y,z)}$ , $q_{d_{(x,y,z)}}$ , $q_{dd_{(x,y,z)}}$ , $q_{-q(s,x,y,z)}$ , $q_{d_{-w(x,y,z)}}$ , $q_{dd_{-w(x,y,z)}}$ . $q_{-q(s,x,y,z)}$ is the rotation of the joint, expresses as a quaternion, while $q_{d_{-w(x,y,z)}}$ is the rotational velocity of the joint.
FloatingPlanarJoint(String jname, Robot rob) FloatingPlanarJoint(String jname, Robot rob, int type)	Creates a 3 degree of freedom planar joint. This joint can only be the root joint for a robot. YoVariables $q_{(t1,t2,rot)}$ , $q_{d_{(t1,t2,rot)}}$ , $q_{dd_{(t1,t2,rot)}}$ and $\tau_{(t1,t2,rot)}$ will be automatically created and added to the robot, where $t1$ , $t2$ , are replaced with $x,y$ , or $z$ ; and $rot$ is replaced with $roll$ , $pitch$ , or $yaw$ depending if the axis

	of rotation is x,y,or z. The type is one of XY, XZ, or YZ which defines the plane of motion. If no type is selected, then XZ is the default.
public static final int XY; public static final int YZ; public static final int XZ;	Constant parameters for joint type selection when creating a PlanarFloatingJoint.
PinJoint(String jname, Vector3d offset, Robot rob, int jaxis)	Creates a pin joint with variables q_*, qd_*, qdd_*, tau_* with * replaced by the joint name. Rotation is about jaxis which can be Joint.X, Joint.Y, or Joint.Z. offset is the Vector3d from the previous joint.
PinJoint(String jname, Vector3d offset, Robot rob, Vector3d jaxis)	Creates a pin joint as above, but with the Joint Axis pointing along the Vector3d jaxis.
void setInitialState(double q_init, double qd_init)	Sets the position and velocity of this joint.
void setDamping(double b_damp)	Adds viscous damping to the pin joint.
void setLimitStops(double q_min, double q_max, double k_limit, double b_limit)	Adds limit stops to the pin joint. These limit stops are implemented as a spring-damper system with spring constant k_limit and damper constant b_limit.
SliderJoint(String jname, Vector3d offset, Robot rob, int jaxis)	Creates a slider joint with variables q_*, qd_*, qdd_*, tau_* with * replaced by the joint name. Translation is about jaxis which can be Joint.X, Joint.Y, or Joint.Z. offset is the Vector3d from the previous joint.
SliderJoint(String jname, Vector3d offset, Robot rob, Vector3d jaxis)	Creates a slider joint as above, but with the Joint Axis pointing along the Vector3d jaxis.
void setInitialState(double q_init, double qd_init)	Sets the position and velocity of this joint.
void setDamping(double b_damp)	Adds viscous damping to the slider joint.
void setLimitStops(double q_min, double q_max, double k_limit, double b_limit)	Adds limit stops to the slider joint. These limit stops are implemented as a spring-damper system with spring constant k_limit and damper constant b_limit.
CylinderJoint(String rotName, String transName, Vector3d offset, Robot rob, int jaxis)	Creates a cylinder joint, which is a pin joint plus a slider joint. Both rotation and translation are about the same axis, which can be Joint.X, Joint.Y, or Joint.Z.
void setInitialState(double q1_init, double qd1_init, double q2_init, double qd2_init)	Sets the positions and velocities of this joint.
void setDamping(int axis, double b_damp)	Adds damping to the indicated axis of the CylinderJoint (1 = pin, 2 = slider).
void setLimitStops(int axis, double q_min, double q_max, double k_limit, double b_limit)	Adds limit stops to the indicated axis of the CylinderJoint.
UniversalJoint(String jname1, String jname2, Vector3d offset, Robot rob, int firstAxis, int secondAxis)	Creates a universal joint, which is two pin joints in a row. Rotations are about firstAxis and secondAxis, which can be Joint.X, Join.Y, or Joint.Z. The same axis cannot be used for both joints however.
void setInitialState(double q1_init, double qd1_init, double q2_init, double qd2_init)	Sets the positions and velocities of this joint.

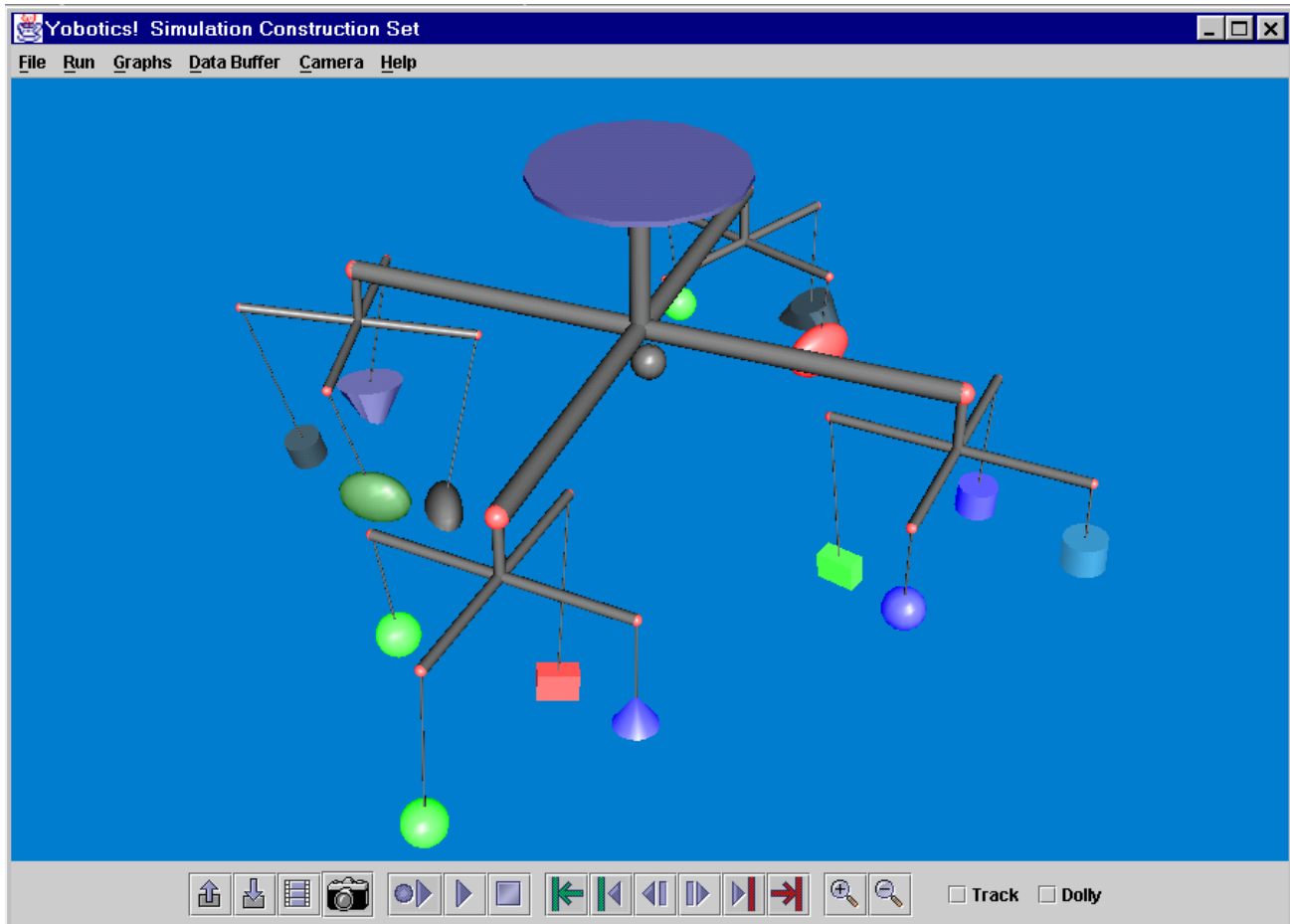
<code>void setDamping(int axis, double b_damp)</code>	Adds damping to the indicated axis of the UniversalJoint (1 = firstAxis, 2 = secondAxis).
<code>void setLimitStops(int axis, double q_min, double q_max, double k_limit, double b_limit)</code>	Adds limit stops to the indicated axis of the UniversalJoint.
<code>GimbalJoint(String jname1, String jname2, String jname3, Vector3d offset, Robot rob, int firstAxis, int secondAxis, int thirdAxis)</code>	Creates a gimbal joint, which is three pin joints in a row. The axis of rotation must all be perpendicular (one each of Joint.X, Joint.Y, and Joint.Z)
<code>void setInitialState(double q1_init, double qd1_init, double q2_init, double qd2_init, double q3_init, double qd3_init)</code>	Sets the positions and velocities of this joint.
<code>void setDamping(int axis, double b_damp)</code>	Adds damping to the indicated axis of the GimbalJoint (1 = firstAxis, 2 = secondAxis, 3 = thirdAxis).
<code>void setLimitStops(int axis, double q_min, double q_max, double k_limit, double b_limit)</code>	Adds limit stops to the indicated axis of the GimbalJoint.

**Table 7: General Joint Methods.**

Method or Constructor	Purpose
<code>void setLink(Link l)</code>	Sets the link for this joint.
<code>void addJoint (Joint nextJoint)</code>	Adds a child joint to this joint.
<code>void addKinematic (KinematicPoint point)</code>	Adds a child kinematic point to this joint.
<code>void addExternalForcePoint (ExternalForcePoint point)</code>	Adds a child external force point to this joint.
<code>void addGroundContactPoint (GroundContactPoint point)</code>	Adds a child ground contact point to this joint.
<code>void changeOffsetVector(Vector3d newOffsetVector)</code> <code>void changeOffsetVector(double x, double y, double z)</code>	Changes the offset Vector for this Joint. Should only be used to change a robot's shape (and not as a way to actuate a degree of freedom), since the use of this function does not obey the laws of physics.
<code>public static final int Joint.X;</code> <code>public static final int Joint.Y;</code> <code>public static final int Joint.Z;</code>	Constants for specifying the joint axis in the joint constructors.

## Tutorial 5: Creating a Robot With Multiple Joints.

In this tutorial, we will create a simulation of a child's mobile toy, using a tree structure of 21 gimbal joints (63 degrees of freedom total). Figure 6 shows a screen shot of the mobile simulation. In this tutorial you will gain more experience with specifying joint offsets, and link translations and rotations.



**Figure 6: MobileSimulation.** The mobile consists of 21 gimbal joints, 63 degrees of freedom total.

1. Create a new project in JBuilder with the name Mobile.
2. Add the class MobileSimulation and fill it in as follows:

```
package com.yobotics.exampleSimulations.mobile;  
  
import com.yobotics.simulationconstructionset.*;  
  
public class MobileSimulation  
{  
    private SimulationConstructionSet sim;
```

```

public MobileSimulation()
{
    MobileRobot mobile = new MobileRobot();

    sim = new SimulationConstructionSet(mobile);
    sim.setGroundVisible(false);

    sim.setCameraTracking(false, false, false, false);
    sim.setCameraDolly(false, false, false, false);

    sim.setCameraPosition(1.0, 1.0, 0.5);
    sim.setCameraFix(0.0, 0.0, 0.8);

    sim.setCameraTrackingVars("ef_track00_x", "ef_track00_y", "ef_track00_z");

    sim.setDT(0.02, 1);
    Thread myThread = new Thread(sim);
    myThread.start();
}

public static void main(String[] args)
{
    MobileSimulation sim = new MobileSimulation();
}
}

```

3. Note that the integration time step is set to 0.02 seconds: `sim.setDT(0.02, 1);`. This is a fairly large time step, but in this case is possible since there are no high-frequency interactions in the system. In general, to set the integration time step, experiment with different values. As the time step gets higher, you will see different behavior due to numerical instabilities. Choose a time step which is low enough that making it any lower does not change the outcome of the simulation. The second parameter to `setDT` is 1, meaning that every integration step will be recorded.
4. Note that the ground is set to be invisible, camera tracking and dollying are turned off, and that the camera position and fix are set to get a good view of the mobile:
  - a. `sim.setGroundVisible(false);`
  - b. `sim.setCameraTracking(false,false,false,false);`
  - c. `sim.setCameraDolly(false,false,false,false);`
  - d. `sim.setCameraPosition(1.0,1.0,0.5);`
  - e. `sim.setCameraFix(0.0,0.0,0.8);`
5. Add the class `MobileRobot` and fill it in as follows:

```

package com.yobotics.exampleSimulations.mobile;

import com.yobotics.simulationconstructionset.*;
import javax.vecmath.*;
import javax.media.j3d.Appearance;

public class MobileRobot extends Robot
{
    private static final double

```

```

    L1 = 0.3, M1 = 0.1, R1 = 0.01, Ixx1 = 0.01, Iyy1 = 0.01, Izz1 = 0.01;
private static final double
    L2 = 0.12, M2 = 0.05, R2 = 0.005, Ixx2 = 0.01, Iyy2 = 0.01, Izz2 = 0.01;
private static final double
    L3 = 0.08, M3 = 0.03, R3 = 0.001, Ixx3 = 0.01, Iyy3 = 0.01, Izz3 = 0.01;;
private static final double
    TOY_L = 0.02, TOY_W = 0.04, TOY_H = 0.03, TOY_R = 0.02;

private static final double
    DAMP1 = 0.06, DAMP2 = 0.006, DAMP3 = 0.003;

public MobileRobot()
{
    super("Mobile");

    Link topLink = new Link("top");
    topLink.translate(0.0, 0.0, 1.0 + R1 / 2.0);
    topLink.addCylinder(L1 / 60.0, L1 / 3.0, YoAppearance.DarkBlue());
    this.addStaticLink(topLink);

    GimbalJoint firstGimbal = new GimbalJoint("gimbal_x", "gimbal_y",
"gimbal_z", new Vector3d(0.0, 0.0, 1.0), this, Joint.X, Joint.Y, Joint.Z);
    Link bar1 = crossBar(M1, L1, R1, Ixx1, Iyy1, Izz1);
    firstGimbal.setLink(bar1);
    firstGimbal.setDamping(DAMP1);
    initJoint(firstGimbal);

    this.addRootJoint(firstGimbal);

    double xOffset, yOffset;
    GimbalJoint nextGimbal;
    GimbalJoint finalGimbal;
    Link nextLink;

    for (int i = 0; i < 4; i++)
    {
        xOffset = 0.0;
        yOffset = 0.0;

        if (i == 0)
            xOffset = L1;
        if (i == 1)
            xOffset = -L1;
        if (i == 2)
            yOffset = L1;
        if (i == 3)
            yOffset = -L1;

        nextGimbal = new GimbalJoint("gimball_" + i + "_x", "gimball_" + i +
            "_y", "gimball_" + i + "_z", new Vector3d(xOffset, yOffset, -L1 / 2.0), this,
                Joint.X, Joint.Y, Joint.Z);
        nextLink = crossBar(M2, L2, R2, Ixx2, Iyy2, Izz2);
        nextGimbal.setLink(nextLink);
        nextGimbal.setDamping(DAMP2);
        initJoint(nextGimbal);
        firstGimbal.addJoint(nextGimbal);
    }
}

```

```

    for (int j = 0; j < 4; j++)
    {
        xOffset = 0.0;
        yOffset = 0.0;

        if (j == 0)
            xOffset = L2;
        if (j == 1)
            xOffset = -L2;
        if (j == 2)
            yOffset = L2;
        if (j == 3)
            yOffset = -L2;

        finalGimbal = new GimbalJoint("gimbal2_" + i + "_" + j + "_x",
"gimbal2_" + i + "_" + j + "_y", "gimbal2_" + i + "_" + j + "_z", new
Vector3d(xOffset, yOffset, -L2 / 2.0), this, Joint.X, Joint.Y, Joint.Z);
        nextLink = randomShape();
        finalGimbal.setLink(nextLink);

        ExternalForcePoint point = new ExternalForcePoint("ef_track" + i +
j, new Vector3d(0.0, 0.0, -2.0 * L3), this);
        finalGimbal.addExternalForcePoint(point);

        finalGimbal.setDamping(DAMP3);
        initJoint(finalGimbal);
        nextGimbal.addJoint(finalGimbal);
    }
}
}

private void initJoint(GimbalJoint joint)
{
    double init_q1 = (2.0 * Math.random() - 1.0) * 0.25;
    double init_q2 = (2.0 * Math.random() - 1.0) * 0.25;
    double init_q3 = (2.0 * Math.random() - 1.0) * Math.PI;
    double init_qd1 = (2.0 * Math.random() - 1.0) * 0.5;
    double init_qd2 = (2.0 * Math.random() - 1.0) * 0.5;
    double init_qd3 = (2.0 * Math.random() - 1.0) * 2.0;

    joint.setInitialState(init_q1, init_qd1, init_q2, init_qd2, init_q3,
init_qd3);
}

private Link crossBar(double mass, double length, double radius, double Ixx,
double Iyy, double Izz)
{
    Link ret = new Link("CrossBar");
    ret.setMass(mass);
    ret.setComOffset(0.0, 0.0, -length / 2.0);
    ret.setMomentOfInertia(Ixx, Iyy, Izz);

    ret.addSphere(R1, YoAppearance.Red());
    ret.translate(0.0, 0.0, -length / 2.0);
}

```

```

ret.addCylinder(length / 2.0, radius);

ret.identity();
ret.translate(length, 0.0, -length / 2.0);
ret.rotate(-Math.PI / 2.0, Link.Y);
ret.addCylinder(2.0 * length, radius);
ret.addSphere(radius, YoAppearance.Red());
ret.translate(0.0, 0.0, 2.0 * length);
ret.addSphere(radius, YoAppearance.Red());

ret.identity();
ret.translate(0.0, length, -length / 2.0);
ret.rotate(Math.PI / 2.0, Link.X);
ret.addCylinder(2.0 * length, radius);
ret.addSphere(radius, YoAppearance.Red());
ret.translate(0.0, 0.0, 2.0 * length);
ret.addSphere(radius, YoAppearance.Red());

return ret;
}

private Link randomShape()
{
    Link ret = new Link("randomShape");
    double stringLength = L3 * (1.0 + 2.0 * Math.random());

    ret.setMass(M3);
    ret.setComOffset(0.0, 0.0, -stringLength);
    ret.setMomentOfInertia(Ixx3, Iyy3, Izz3);

    ret.translate(0.0, 0.0, -stringLength);
    ret.addCylinder(stringLength, R3);

    Appearance app = YoAppearance.Black();

    int appSelection = (int) (Math.random() * 9.0);
    switch (appSelection)
    {
        case 0 :
            app = YoAppearance.Black();

            break;

        case 1 :
            app = YoAppearance.Red();

            break;

        case 2 :
            app = YoAppearance.DarkRed();

            break;

        case 3 :
            app = YoAppearance.Green();

            break;
    }
}

```

```

case 4 :
    app = YoAppearance.DarkGreen();

    break;

case 5 :
    app = YoAppearance.Blue();

    break;

case 6 :
    app = YoAppearance.DarkBlue();

    break;

case 7 :
    app = YoAppearance.AluminumMaterial();

    break;

case 8 :
    app = YoAppearance.BlackMetalMaterial();

    break;
}

int toySelection = (int) (Math.random() * 7.0);
switch (toySelection)
{
case 0 :
    ret.addSphere(TOY_R, app);

    break;

case 1 :
    ret.addCylinder(TOY_H, TOY_R, app);

    break;

case 2 :
    ret.addCube(TOY_L, TOY_W, TOY_H, app);

    break;

case 3 :
    ret.addCone(TOY_H, TOY_R, app);

    break;

case 4 :
    ret.addEllipsoid(TOY_L, TOY_W, TOY_H, app);

    break;

case 5 :
    ret.addHemiEllipsoid(TOY_L, TOY_W, TOY_H, app);

```

```

        break;

    case 6 :
        ret.addGenTruncatedCone(TOY_H, TOY_L, TOY_W, TOY_W, TOY_L, app);
    }

    return ret;
}
}

```

6. Examine the lines where the 3 different levels of gimbal joints are defined. Note how in Java, Strings can be concatenated using +, and that integers will be turned into Strings in this way. For example, if  $i=3$ , then "gimball1\_"+ $i$ +"\_x" will result in "gimball1\_3\_x". Look at the offset Vector3d for each joint level. The first is (0.0,0.0,1.0). The second is (xOffset, yOffset, -L1/2.0) where (xOffset,yOffset) is either (L1,0), (-L1,0), (0,L1), or (0,-L1). The third is (xOffset, yOffset, -L2/2.0) where (xOffset,yOffset) is either (L2,0), (-L2,0), (0,L2), or (0,-L2). The first offset is (0.0,0.0,1.0) simply to place the mobile into the air. The second offset is the offset from the second level of 4 gimbal joints to each of their parent joint, which is the first gimbal joint. The top crossbar goes down L1/2.0 and in each of the four directions by L1. Therefore, the given offsets should work. The third offset is the offset of the final level of 16 gimbals from their parents above them. The smaller crossbars go down L2/2.0 and in each of the four directions by L2. Therefore, the given offsets should work. Note that the joint offsets are completely independent of any translations or rotations which are performed while specifying links:
  - a. `firstGimbal = new GimbalJoint("gimbal_x", "gimbal_y","gimbal_z", new Vector3d(0.0,0.0,1.0), this, Joint.X, Joint.Y, Joint.Z);`
  - b. `nextGimbal = new GimbalJoint("gimball1_"+ $i$ +"_x", "gimball1_"+ $i$ +"_y", "gimball1_"+ $i$ +"_z", new Vector3d(xOffset, yOffset, -L1/2.0), this, Joint.X, Joint.Y, Joint.Z);`
  - c. `finalGimbal = new GimbalJoint("gimbal2_" +  $i$  + "_" +  $j$  + "_x", "gimbal2_" +  $i$  + "_" +  $j$  + "_y", "gimbal2_" +  $i$  + "_" +  $j$  + "_z", new Vector3d(xOffset, yOffset, -L2 / 2.0), this, Joint.X, Joint.Y, Joint.Z);`
7. Note that viscous damping is set for each joint. Viscous damping of the form  $\tau = \text{damping} * \text{velocity}$  will be added on top of any torques due to joint limits or a control system. For this simulation, the mobile is completely passive and damping is the only source of joint torque:
  - a. `firstGimbal.setDamping(DAMP1);`
  - b. `nextGimbal.setDamping(DAMP2);`
  - c. `finalGimbal.setDamping(DAMP3);`
8. Note that the function `initJoint` is called for each joint. `initJoint` creates random values for the position and velocity of each of the 3 degrees of freedom for the `GimbalJoint` and sets the state using `GimbalJoint.setInitialState()`:
  - a. `joint.setInitialState(init_q1, init_qd1, init_q2, init_qd2, init_q3, init_qd3);`
9. Creation of the joints, including their offsets, damping, and initialization should be clear. If not, review the above joint generation. Note that creation of the joints is completely independent of the creation of the link shapes, and that the joint locations and functions will be the same no matter which links are attached to them.

10. Now let's examine the creation of the links. We start with the top link, which is a flattened cylinder. We add it to the robot using `Robot.addStaticLink(Link)` since it is not attached to any joints and therefore cannot move. Since the first joint of the robot had an offset of `(0.0,0.0,1.0)`, we need to translate to `(0.0,0.0,1.0+R1/2.0)` before adding this link:
- `Link topLink = new Link("top");`
  - `topLink.translate(0.0,0.0,1.0+R1/2.0);`
  - `topLink.addCylinder(L1/60.0,L1/3.0,YoAppearance.DarkBlue());`
  - `this.addStaticLink(topLink);`
11. In creating the crossbar geometry, we first create the upper sphere-capped cylinder which projects downward. Since the origin of a cylinder is the center of its base, we must first translate down the length of the upper cylinder (`length/2.0`) before adding it:
- `ret.addSphere(R1,YoAppearance.Red());`
  - `ret.translate(0.0,0.0,-length/2.0);`
  - `ret.addCylinder(length/2.0, radius);`
12. Next, we create the 2 horizontal cylinders, capped with red spheres on both ends. They both are created identically, except that they are translated and rotated differently. The first one is created as follows:
- `ret.identity();` This line brings us back to the gimbal joint.
  - `ret.translate(length,0.0,-length/2.0);` Translate down `length/2.0` and in the X direction by `length`.
  - `ret.rotate(-Math.PI/2.0, Link.Y);` Rotate by `-Math.PI/2.0` about the Y axis. This will now make the Z axis point in the direction in which we wish to place our cylinder (along the original X axis).
  - `ret.addCylinder(2.0*length,radius);` Add the cylinder.
  - `ret.addSphere(radius,YoAppearance.Red());` Add a sphere to cap this end of the cylinder.
  - `ret.translate(0.0,0.0,2.0*length);` Translate along the Z axis to the other end of the cylinder.
  - `ret.addSphere(radius,YoAppearance.Red());` Cap the other end of the cylinder.
13. The second cylinder is identical to the first, except for the initial translation and rotation:
- `ret.translate(0.0,length,-length/2.0);` Translate down `length/2.0` and in the Y direction by `length`.
  - `ret.rotate(Math.PI/2.0, Link.X);` Rotate by `Math.PI/2.0` about the X axis. This will now make the Z axis point in the direction in which we wish to place our cylinder (along the original Y axis).
14. Examine the `randomShape` code. It will randomly select a string length, an appearance and a shape for each of the 16 toys at the ends of the mobile. How this code works should now be clear to you. If not, experiment with it, or try making a simulation of a different style of mobile to become comfortable with joints and links.

## Kinematic, ExternalForce and GroundContactPoint API

Sometimes you wish to monitor the Cartesian position and velocity of a point on your robot, apply an external force to that point, or model ground contact at that point. The first task can be accomplished using a `KinematicPoint`. The second can be accomplished using an `ExternalForcePoint` which extends `KinematicPoint`. The third can be accomplished using a `GroundContactPoint` which extends `ExternalForcePoint`. Creation of a point requires a name, offset, and robot. The offset is the vector from the joint that the point will be attached to using the `addKinematicPoint`, `addExternalForcePoint`, or `addGroundContactPoint` methods of `Joint`. The point will move with the `Joint` in the same way that a `Link` will move with the `Joint`.

See Table 8 for the `KinematicPoint`, `ExternalForcePoint` and `GroundContactPoint` constructors and methods. When a `KinematicPoint` is created, the `YoVariables` “name\_x”, “name\_y”, “name\_z”, “name\_dx”, “name\_dy”, “name\_dz” are automatically created. These are the position and velocity (in world coordinates) variables for the point. These variables get updated whenever the robot is updated.

The class `ExternalForcePoint` is an extension of `KinematicPoint`. When an `ExternalForcePoint` is created, the `YoVariables` “name\_fx”, “name\_fy”, “name\_fz” are automatically created. These are the forces acting on that point (in world coordinates). These variables can be set by the user to simulated a force acting on the robot at that point, or can be set by a ground contact model, or other object if the `ExternalForcePoint` is being used to model a contact or other event.

The class `GroundContactPoint` is an extension of `ExternalForcePoint`. `GroundContactPoints` also have the `YoVariables` `tdx`, `tdy`, `tdz` which is the location where the `GroundContactPoint` first contacts the ground and `fs`, which is a footswitch. If `fs.val` equals 1.0, then the `GroundContactPoint` is contacting the ground, if it equals 0.0, the point is not contacting the ground. To turn the `GroundContactPoint` off, set `fs.val` equal to -1.0.

Given a `KinematicPoint`, `ExternalForcePoint`, or `GroundContactPoint`, you can extract the automatically created variables. For example the following code will create an `ExternalForcePoint`, attach it to a `Joint`, and apply a force of 1.0 in the Z direction:

```
point1 = ExternalForcePoint("efp1", new Vector3d(), rob);
previousJoint.addExternalForcePoint(point1);
YoVariable efp1_fz = point1.fz;
efp1_fz.val = 1.0;
```

The forces applied to a `GroundContactPoint` are determined by the `GroundContactModel` which is set for the `Robot`. `GroundContactModel` is an interface with the required methods `doGroundContact`, `setGroundProfile`, and `getGroundProfile`. The code for implementing your ground contact model should be placed in `doGroundContact`. There are also a couple `GroundContactModels` that are in the `com.yobotics.simulationconstructionset.utils` package.

A `GroundProfile` defines the contour of your terrain. `GroundProfile` is an interface with the required methods shown in Table 10. By setting a `GroundProfile` to your `GroundContactModel`, and by setting your `GroundContactModel` to your robot, the ground when the simulation is run will be

drawn as the ground profile. The user can generate his/her own GroundProfile, or can use one of the GroundProfiles provided in the com.yobotics.simulationconstructionset.utils package. If no GroundProfile is specified, then a flat ground will be simulated.

**Table 8: ExternalForcePoint and GroundContactPoint constructors and methods.**

Method or Constructor	Purpose
ExternalForcePoint(String name, Vector3d offset, Robot rob)	Creates an external force point, automatically creating the variables name_(x,y,z,dx,dy,dz) and adding them to the given Robot. offset is the Vector3d from the parent joint.
public YoVariable x, y, z; public YoVariable dx, dy, dz;	YoVariables for the cartesian position, and velocity of the kinematic point. These values are automatically computed by the simulation.
ExternalForcePoint(String name, Vector3d offset, Robot rob)	Creates an external force point, which is an extension of a kinematic point. In addition to the kinematic point variables, name_(fx,fy,fz) are automatically generated and added to the Robot.
public YoVariable fx, fy, fz;	YoVariables for the force on the external force point. Force is input by the user, or automatically computed from another object such as a GroundContacModel.
GroundContactPoint(String name, Vector3d offset, Robot rob)	Creates a ground contact point, which is an extension of an external force point. In addition to the external force point variables, name_(tdx,tdy,tdz,fs) are automatically generated and added to the Robot.
public YoVariable tdx, tdy, tdz; public YoVariable fs;	Touch-down position and footswitch. fs.val = 1.0 during foot contact and 0.0 otherwise. To turn the ground contact point off, set fs.val to -1.0.

**Table 9: GroundContactModel Interface. All GroundContactModels must implement this interface.**

Method or Constructor	Purpose
public abstract void doGroundContact();	Method for setting the ground contact forces given their positions, velocities, and the ground profile.
public abstract void setGroundProfile(GroundProfile profile); public abstract GroundProfile getGroundProfile();	Sets and returns the GroundProfile.

**Table 10: GroundProfile Interface. All GroundProfiles must implement this interface.**

Method or Constructor	Purpose
public abstract double heightAt(double x, double y, double z);	Returns the height of this ground profile, given the x, y, and z values.
public abstract boolean isClose(double x, double y, double z);	Returns true if the ground contact point has penetrated this ground profile. Otherwise returns either true or false. Used only for efficiency, so it is always safe to return true.
public abstract void closestIntersectionTo(double x, double y, double z, Point3d intersection);	Sets the point intersection to be the closest intersection between the ground profile and the point at (x,y,z). If computation of the intersection is too difficult, return (x, y, heightAt(x,y,z)).
public abstract void surfaceNormalAt(double x, double y, double z, Vector3d normal);	Sets the vector normal to be the surface normal of the ground at point (x,y,z). Straight up is the vector (0,0,1)
public abstract void closestIntersectionAndNormalAt(double x, double y, double z, Point3d intersection, Vector3d normal);	Sets both the point intersection and surface normal. This method is redundant, but exists due to potential efficiency improvements by computing the intersection and surface normal at the same time. It is safe to implement this method by calling both closestIntersectionTo() and surfaceNormalAt()
public abstract double getXMin(); public abstract double getXMax(); public abstract double getYMin(); public abstract double getYMax();	Returns the boundaries of the ground profile. Everything outside this bound will be drawn at a height of zero. Inside this bound, the ground will be drawn as per the profile. The smaller the bound, the more detailed the graphics. This bound does not effect the resolution of the ground physics, however, which is only bounded by the precision of the numbers.
public abstract double getXTiles(); public abstract double getYTiles();	Returns the number of tilings to perform in both the x and y direction if the ground is texture mapped.

## Tutorial 6: Ground Contact Modeling.

1. Run the simulation in ExampleSimulations\FallingBrick
2. Examine the source code. Find the following 4 lines in FallingBrickRobot where the ground contact is defined:
  - a. `GroundContactModel groundModel = new LinearGroundContactModel(this, 1422, 150.6, 50.0, 1000.0);`
  - b. `GroundProfile profile = new WavyGroundProfile();`
  - c. `groundModel.setGroundProfile(profile);`
  - d. `this.setGroundContactModel(groundModel);`
3. We see that the groundModel is an instance of the class LinearGroundContactModel and the GroundProfile is an instance of the class WavyGroundProfile(). Lets look at one at a time.
4. Examine the file WavyGroundProfile.java. Since WavyGroundProfile implements the interface GroundProfile, it defines the methods heightAt, isClose, surfaceNormalAt, closestIntersectionTo, closestIntersectionAndNormalAt, getXmin, getXmax, getYmin, getYmax, getXTiles, getYTiles. The point where the height of the ground is defined is in heightAt:

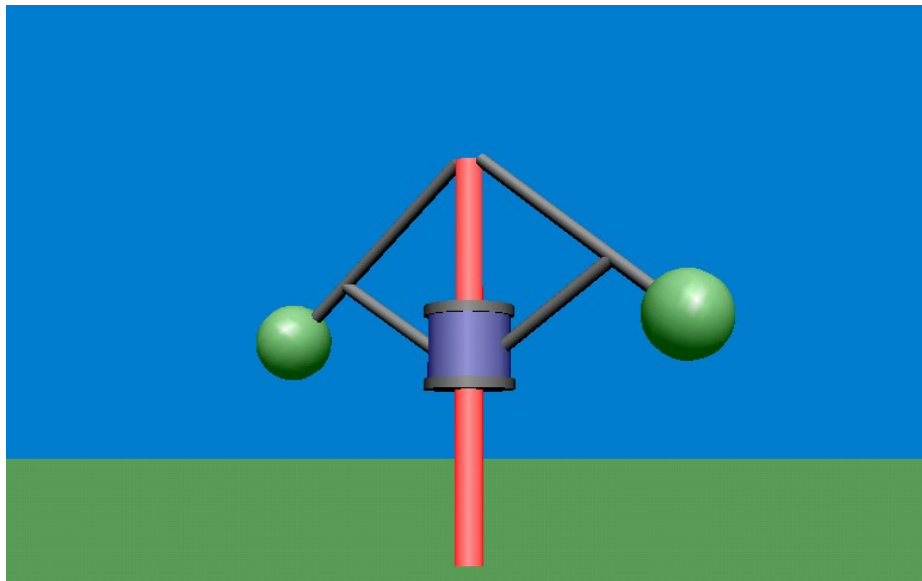
```
return 1.0 * Math.exp(-Math.abs(2.0*x)) * Math.exp(-Math.abs(2.0*y)) *  
        Math.sin(2.0*Math.PI*0.7*x);
```
5. Change the profile of the terrain by changing the function heightAt. Run the simulation and see how the profile of the ground changed.
6. Examine the file LinearGroundContactModel.java. Since LinearGroundContactModel implements the interface GroundContactModel, it defines the method doGroundContact, where the ground contact forces are computed.
7. Study the doGroundContact method. Notice that the ground is modeled as a linear spring-damper in the x and y directions and a non-linear spring and a linear damper in the z direction. Using a non-linear (hardening) spring in the z direction is a standard way to prevent ground chatter or bounce while still simulating a stiff ground. This GroundContactModel is a simple one and does not take into consideration the surface normal of the ground, or ground slipping. For this example simulation, that is ok, but in many instances, these effects are important. If so, then you should use one of the other ground contact models in the com.yobotics.simulationconstructionset.utils package, or create your own.
8. Experiment with different values of ground spring and damping constants. Note that as you lower the stiffnesses, greater penetration into the ground will occur. As you lower the damping constants, vibrations occur longer. However, if you increase the stiffness or damping constants too much, instabilities can occur due to numerical instabilities.
9. As a general rule of thumb, ground stiffnesses and damping are tuned experimentally until an acceptable ground penetration and bounce is achieved. Try tuning the ground parameters for different ground penetration and bounce.

Various GroundProfiles and GroundContactModels are provided in the com.yobotics.simulationconstructionset.utils directory. You should examine the source code of these files (found in SimulationConstructionSet\src) in order to determine the model and profile best for your simulation, and to understand how to create your own custom ground contact model.

## Tutorial 7: Implementing Closed-Chain Mechanisms Using External Force Points

Currently, the Simulation Construction Set only allows for systems which have a tree structure of Joints and Links. However, Closed-Chain Mechanisms, like 4 bar linkages for example, can be implemented using ExternalForcePoints. Two ExternalForcePoints can be placed at two different points on the system which are constrained to stay together. A “glue” force can then be computed based on the error in position between the two points. That glue force can then be applied to the ExternalForcePoints in equal and opposite directions.

In this tutorial, you’ll see how to construct a Flyball Governor simulation, like the one shown in Figure 7. Since this is a closed-loop mechanism, we use ExternalForcePoints to hold the cross links to the rotating blue cylinder.



**Figure 7: Flyball Governor Simulation. External Force Points are used to implement a closed-loop chain mechanism.**

1. Run JBuider and create the project FlyballGovernor using the existing source in SimulationConstructionSet/ExampleSimulations/FlyballGovernor.
2. Run the simulation. Note that the blue cylinder rises as the device spins faster. To vary the desired speed, change the value of `q_d_cylinder_z`.
3. Examine the FlyballGovernorRobot source code. Note where the ExternalForcePoints are created and attached.
4. Examine `doControl()`. Here is where the constraint forces are generated. We see that for each constraint, a linear spring-damper is used to “glue” the two ExternalForcePoints together:

```
constraint1A_fx.val = K * (constraint1B_x.val - constraint1A_x.val)
                    + B * (constraint1B_dx.val - constraint1A_dx.val);
constraint1A_fy.val = K * (constraint1B_y.val - constraint1A_y.val)
```

```

        + B * (constraint1B_dy.val - constraint1A_dy.val);
constraint1A_fz.val = K * (constraint1B_z.val - constraint1A_z.val)
        + B * (constraint1B_dz.val - constraint1A_dz.val);

constraint1B_fx.val = -constraint1A_fx.val;
constraint1B_fy.val = -constraint1A_fy.val;
constraint1B_fz.val = -constraint1A_fz.val;

```

5. The last line of doControl is the feedback mechanism used in the FlyballGovernor:
 

```
tau_rotation.val = k_feedback.val * (q_d_cylinder_z.val - q_cylinder_z.val);
```

 The torque on the rotation joint is proportional to the position of the cylinder. In a real implementation, such as on a locomotive, this would be implemented by mechanically coupling the cylinder to the throttle. In essence, a FlyballGovernor provides velocity sensing, feedback, and control completely mechanically.
6. Change the value of K and B to 0.0 and rerun the simulation. See that the constraint is no longer enforced and the “glue” joint comes apart.
7. Try implementing a closed-loop mechanism on your own. Examples include four-bar linkages or a necklace with rigid links.

## SimulationConstructionSet API

The following tables list the API for the SimulationConstructionSet class. These methods are usually called when creating a simulation and are for setting the parameters of the simulation, such as the integration time step, the camera position and fix, and for specifying which variables are initially plotted in the graphs.

There are two constructors for a SimulationConstructionSet. Both require a Robot, rob. In the second constructor, if boolean showGUI equals false, then no Graphical User Interface will be displayed. This is useful when you wish to run a simulation, or several simulations quickly, but do not need to interact through a GUI. In both cases, before the simulation can be simulated, you must first create a Thread, given the SimulationConstructionSet, and start the Thread.

To start simulating when not using a GUI, use the simulate() functions. To know when the simulation is finished, use addSimulateDoneListener(). SimulateDoneListener is an interface that contains the single method simulateDone(). To stop the simulation, use stop().

Method or Constructor	Purpose
SimulationConstructionSet(Robot rob) SimulationConstructionSet(Robot rob, boolean showGUI)	Creates a new simulation, given a robot. Creates a new simulation, If showGUI is false, then no GUI appears.
void setDT(double simulateDT, int recordFrequency) double getDT()	Sets the integration time step and variable record frequency (steps per record). Defaults are 0.0004 seconds and 50 steps per record (0.02 seconds per record).
void setPlaybackRealTimeRate(double rate)	Sets the playback slow-motion/fast-motion rate. The default value of 1.0 means real time playback.
void setPlaybackDesiredFrameRate(double rate)	Sets the desired graphics update rate during playback. Default value is 0.04, or 25 fps.
YoVariable getVar(String varname)	Finds and returns a YoVariable of the given name.
void addStaticLink(Link staticLink)	Adds a static link with no physics. For creating the surrounding static world.
public void setFastSimulate(boolean fastSimulate)	If fastSimulate is true, makes the GUI less responsive during simulation in order to increase simulation speed.
void addStaticBranchGroup(BranchGroup staticBranchGroup)	Adds a static branch group with no physics. For creating the surrounding static world.

Method or Constructor	Purpose
void setupVarGroup(String name, String[] varNames) void setupVarGroup(String name, String[] varNames, String[] regularExpressions)	Defines a group of variables that can be selected via the GUI. The first String array contains exact names. The second optional String array is regular expressions that will be matched to YoVariable names.
void setupGraphGroup(String name, String[][] varNames)	Defines a graph group that can be selected via the GUI.
void setupEntryBoxGroup(String name, String[] varNames)	Defines a group of entry boxes that can be selected via the GUI.
void setupConfiguration(String name, String varGroup, String graphGroup, String entryBoxGroup)	Defines a configuration consisting of a variable group, a graph group, and an entry box group. Use "all" for the varGroup to display all the variables.
void selectConfiguration(String name)	Selects the default configuration for the GUI.

Method or Constructor	Purpose
public void simulate()	Simulate continuously.
public void simulate(double simulationTime)	Simulate for simulationTime time.
public void simulate(int numTicks)	Simulate for numTicks ticks.
Public void addSimulateDoneListener(SimulateDoneListener listener)	Adds a SimulateDoneListener that is called back when simulate() is completed. SimulateDoneListener is an interface whose sole method is public void simulateDone()
public void stop()	Stop simulating.

Method or Constructor	Purpose
public void setExportDataDirectory(String directory)	Sets the default directory for exporting data.
public void setImportDataDirectory(String directory)	Sets the default directory for importing data.

Method or Constructor	Purpose
void setCameraFix(double fixX, double fixY, double fixZ)	Sets the coordinates of the position that the camera is fixed to (looking at).
void setCameraTracking(boolean track, boolean trackX, boolean trackY, boolean trackZ)	Sets whether the camera is tracking an object or not. Tracking along individual axes (x,y,z) can be set.
void setCameraTrackingOffsets(double dx, double dy, double dz)	Sets the offset vector of the fix point of the camera from the object it is tracking.
void setCameraTrackingVars(String xName, String yName, String zName)	Sets the variable names of the object the camera is tracking. Default is q_x, q_y, q_z.
void setCameraPosition(double posX, double posY, double posZ)	Sets the coordinates of the position of the camera.
void setCameraDolly(boolean dolly, boolean dollyX, boolean dollyY, boolean dollyZ)	Sets whether the camera is dollying relative to an object or not. Dolly along individual axes (x,y,z) can be set.
void setCameraDollyOffsets(double dx, double dy, double dz)	Sets the camera offset vector from the object it is dollying relative to.
void setCameraDollyVars(String xName, String yName, String zName)	Sets the variable names of the object the camera is dollying relative to. Default is q_x, q_y, q_z.

Method or Constructor	Purpose
public void setGroundVisible(boolean isVisible)	Makes the ground invisible if isVisible is false, otherwise makes the ground visible.
void setGroundAppearance(Appearance app)	Sets the Appearance of the ground to app.

Method or Constructor	Purpose
void setupEntryBox(String varname) void setupEntryBox(String[] varnames)	Adds a numeric entry box to the graphical user interface with the indicated variable(s).
void setupGraph(String varname) void setupGraph(String[] varnames)	Adds a graph to the graphical user interface with the indicated variable(s).

Method or Constructor	Purpose
void addButton(JButton button)	Adds a user-created JButton to the graphical user interface.
void addCheckBox(JCheckBox checkBox)	Adds a user-created JCheckBox to the graphical user interface.
void addRadioButton(JRadioButton button)	Adds a user-created JRadioButton to the graphical user interface.

## Technical Support, Bugs and Desired Features

Please send bug reports, requests for desired features, and technical support questions to support@yobotics.com. For urgent technical support call (850)602-5860.

We are interested in hearing about your work. Please send us example simulations so that we can share them with other users. Happy Simulating!

## Getting Started from The Ground Up

This section describes a list of tasks that a person new to Java and the SimulationConstructionSet can perform to get up to speed.

1. Create a new project in JBuilder, or whatever your favorite IDE is. Get up to speed on your IDE by reading it's online help, tutorials, etc. Make sure you are comfortable with making new projects, creating new classes, setting classpaths, compiling and running programs, etc.
2. Create a new class called HelloWorld. Implement a main that simply prints out "HelloWorld".
3. Create some simple programs that iterate over numbers and do some math things. For example:
  - a. Add all the integers from 0 to 1 million.
  - b. Print out the Fibonacci sequence.
  - c. Print out the prime numbers. (Don't worry about efficiency, just use a quick algorithm.)
4. If Java is new to you, read one of the various Java Tutorials available on the internet and do some of the examples in it. As fun as it is, don't worry too much about GUIs/Swing/Java3D until you need to write a GUI.
5. Read up on interfaces and abstract classes in Bruce Eckel's book "Thinking in Java." Do some of the examples in that book.
6. Create an interface of your own and implement two different classes that implement that interface. Choose something conducive to Object Oriented Programming. Potential examples would be
  - a. Polygon which has size, area, numberOfSides, etc. and Triangle/Rectangle that implement Polygon.
  - b. Furniture, Desk/Chair/Table.
  - c. Building, House/Office/FireStation.
  - d. Etc...
7. Read up on ArrayList. You'll use it a lot. Write a program that iterates over an ArrayList<X> where X is the interface you implemented above. For each object in the list, print out the attributes of that object. Also read up on and explore the other containers and write some simple programs using some of them.
8. Implement a class that extends another class. Add some methods to the new class.
9. Implement an enum and iterate over its values, printing them out. Write a switch/case statement using those enums.
10. Implement a program that has a few classes. Avoid having a GUI/Network stuff for now. Focus on classes, interfaces, and simple design. Some possibilities include:

- a. A simple game like TicTacToe. Classes and interfaces could be things like Piece, Board, Move, Player, Referee, etc. Make moves through method calls. Don't worry about GUI stuff.
  - b. A simple ATM program. Classes and interfaces could be things like Account, ATM, Customer, etc. The program could allow checking balances, withdrawing and depositing money, etc.
  - c. A more complex game like checkers or chess. Again, don't worry about GUI stuff. Make moves through function calls.
  - d. A Sudoku solver.
11. Reread "Thinking in Java" while doing the examples. Come up with some of your own related examples. Repeat.

### **Additional Source of Info:**

#### Java Books:

1. Bruce Eckel, "Thinking in Java". Available in bookstores or at [bruceeckel.com](http://bruceeckel.com)
2. Sun, "The Java 3D API Specifications". Available in bookstores or at [java.sun.com](http://java.sun.com)

#### Robotics Books:

1. John Craig, "Introduction to Robotics, Mechanics and Control"

#### Web Sites:

1. [java3d.dev.java.net](http://java3d.dev.java.net), Home of Java3d. Source code, examples, and forums.